

**System Composer™**

Reference



**MATLAB® & SIMULINK®**

R2023a



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



The MathWorks, Inc.  
1 Apple Hill Drive  
Natick, MA 01760-2098

## *System Composer™ Reference*

© COPYRIGHT 2019–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## **Revision History**

March 2019	Online only	New for Version 1.0 (Release 2019a)
September 2019	Online only	Revised for Version 1.1 (Release 2019b)
March 2020	Online only	Revised for Version 1.2 (Release 2020a)
September 2020	Online only	Revised for Version 1.3 (Release 2020b)
March 2021	Online only	Revised for Version 2.0 (Release 2021a)
September 2021	Online only	Revised for Version 2.1 (Release 2021b)
March 2022	Online only	Revised for Version 2.2 (Release 2022a)
September 2022	Online only	Revised for Version 2.3 (Release 2022b)
March 2023	Online only	Revised for Version 2.4 (Release 2023a)

<b>1</b>	<b>Blocks</b>
<b>2</b>	<b>Objects</b>
<b>3</b>	<b>Classes</b>
<b>4</b>	<b>Functions</b>
<b>5</b>	<b>Methods</b>
<b>6</b>	<b>Tools and Apps</b>



# Blocks

---

# Adapter

Connect components with different interfaces

## Description

The Adapter block allows you to connect the source and destination ports of components that have different interface definitions.



To add or connect System Composer components:

- Add an Adapter block from the **Modeling** tab or the palette. The Adapter block has In and Out ports.
- Click and drag a port to create a connection. Connect each port to another component. You can also create a new component to complete the connection.
- Insert an Adapter block between two ports with different interfaces. You can create mappings between interface elements on each port.

To map between interfaces, apply interface conversions, and enter bus creation mode for architecture models:

- Double-click the Adapter block to open the “Interface Adapter” dialog. From here, you can create and edit mappings between input and output interfaces, and set the **Apply Interface conversion** parameter to: `UnitDelay` to break an algebraic loop or `RateTransition` to reconcile different sample time rates for reference models. When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces as you work.

To merge multiple message lines for architecture models and multiple signal or message lines for software architecture models:

- Manually configure the Adapter block by double-clicking the block to open the “Interface Adapter”. Set the **Apply Interface conversion** parameter to `Merge`.
- For software architecture models, from the toolstrip, add a Merge block, which is a preconfigured Adapter block for merging.

## Limitations

- When used for structural interface adaptations, the Adapter block uses bus element ports internally and, subsequently, only supports virtual buses.
- The Adapter block does not support mixing messages and signals as inputs and outputs.

## Ports

### Input

**Source** — Input connection from a component interface

If you connect to a source component, the interfaces on the ports should be compatible.

### Output

**Destination** — Output connection to a component interface

If you connect to a destination component, the interfaces on the ports should be compatible.

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>“Manage Interfaces with Data Dictionaries”</li> <li>“Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>“Create Architecture Model with Interfaces and Requirement Links”</li> <li>“Define Port Interfaces Between Components”</li> </ul>

Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”



Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

#### Functions

connect

#### Blocks

Component | Reference Component | Variant Component

#### Topics

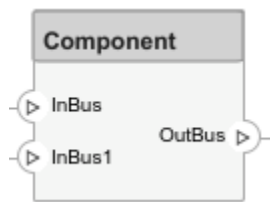
“Define Port Interfaces Between Components”

# Component

Add components to architecture model

## Description

Use a Component block to represent a structural or behavioral element at any level of an architecture model hierarchy. Add ports to the block to connect to other components. Define an interface for the ports and add properties using stereotypes.



To add or connect System Composer components:

- Add an architecture Component block from the **Modeling** tab or the palette. You can also click and drag a box on the canvas, then select the Component block.
- To add a port, select an edge of the component and choose a direction from the menu: **Input**, **Output**, or **Physical**
- Click and drag the port to create a connection. Connect to another component. You can also create a new component to complete the connection.
- To connect Component blocks to architecture or composition model root ports, drag from the component ports to the containing model boundary. When you release the connection, a root port is created at the boundary.
- To add component-level parameters, use the **Parameter Editor** tool.

## Ports

### Input

**Source** — Input connection from another component  
interface

If you connect to a source component, the interfaces on the ports are shared.

### Output

**Destination** — Output connection to another component  
interface

If you connect to a destination component, the interfaces on the ports are shared.

### Physical

**Physical** — Physical connection to another component  
physical interface

If you connect to another component, the physical interfaces on the ports are shared.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
physical subsystem	A physical subsystem is a Simulink® subsystem with Simscape™ connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"

Term	Definition	Application	More Information
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on Ports”
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	“Describe Component Behavior Using Simscape”

## Version History

Introduced in R2019a

## See Also

### Functions

`addComponent` | `addPort` | `connect`

### Blocks

Reference Component | Variant Component | Adapter

### Topics

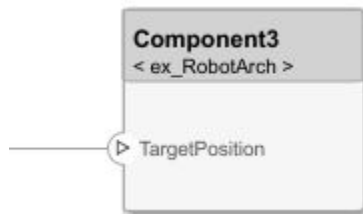
“Compose Architectures Visually”

## Reference Component

Link to architectural definition or Simulink behavior

### Description

Use a Reference Component block to link an architectural definition of a System Composer component or a Simulink behavior.



To add or connect System Composer components:

- Add an architecture Reference Component block from the **Modeling** tab or the palette. You can also click and drag a box on the canvas, then select the Reference Component block.
- Attach a referenced model to the component by selecting <Enter Model Name>.
- Click and drag any port to create a connection. Connect to another component. You can also create a new component to complete the connection.
- To connect Reference Component blocks to architecture or composition model root ports, drag from the component ports to the containing model boundary. When you release the connection, a root port is created at the boundary.

To manage Reference Component block contents:

- When you create a Reference Component block, you have the option to right-click the component and select **Block Parameters**. From here, you can specify your reference model name, if it already exists. The reference model can be a System Composer architecture model or a Simulink model.
- With a regular Component block, you can right-click on the block and convert it to a reference component.
  - Select **Save As Architecture** to save the contents of the component as an architecture model or subsystem that can be referenced in multiple places and kept in sync. The component will become a reference component that links to the referenced architecture model or subsystem.

---

**Note** To type ports on architecture subsystems with interfaces, you must link an external interface data dictionary. Architecture subsystems do not contain a model workspace. For more information, see “Manage Interfaces with Data Dictionaries”.

---

- Select **Create Simulink Behavior** to create a new Simulink reference model or subsystem and link to it.
- Select **Link to Model** to link to a known model or subsystem that can be either a System Composer architecture model or a Simulink model.

- To break the reference link for a Reference Component block, you have the option to right-click and select **Inline Model**, which removes the contents of the architecture model referenced by the specified component and breaks the link to the reference model. The Reference Component block becomes a regular Component block.

---

**Note** Components with physical ports cannot be saved as architecture models, model references, software architectures, or Stateflow® chart behaviors. Components with physical ports can only be saved as subsystem references or subsystem component behaviors.

---

## Ports

### Input

**Source** — Input connection from another component  
interface

If you connect to a source component, the interfaces on the ports are shared.

### Output

**Destination** — Output connection to another component  
interface

If you connect to a destination component, the interfaces on the ports are shared.

### Physical

**Physical** — Physical connection to another component  
physical interface

If you connect to another component, the physical interfaces on the ports are shared.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	<p>Perform operations on a model:</p> <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> <p>A System Composer model is stored as an SLX file.</p>	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”



<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2019a

### See Also

#### Functions

`addComponent` | `addPort` | `connect` | `inlineComponent` | `createSimulinkBehavior` | `createArchitectureModel` | `createStateflowChartBehavior` | `extractArchitectureFromSimulink` | `linkToModel` | `isReference`

#### Blocks

`Component` | `Variant Component` | `Adapter`

#### Topics

“Implement Component Behavior Using Simulink”

“Decompose and Reuse Components”

“Implement Component Behavior Using Stateflow Charts”

“Create Simulink Subsystem Behavior Using Subsystem Component”

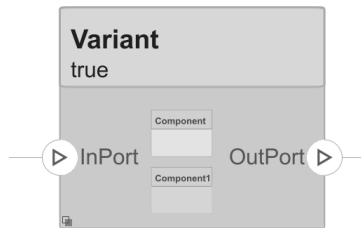
“Simulate and Deploy Software Architectures”

# Variant Component

Add components with alternative designs

## Description

Use a Variant Component block to create multiple design alternatives for a component.



To add or connect System Composer components:

- Add an architecture Variant Component block from the **Modeling** tab or the palette. You can also click and drag a box on the canvas, then select the Variant Component block. You can also create a variant component from a component or reference component. Right-click on the component and select **Add Variant Choice**.
- To add a port, select an edge of the component and choose a direction from the menu: Input or Output
- Click and drag the port to create a connection. Connect to another component. You can also create a new component to complete the connection.
- To connect Variant Component blocks to architecture or composition model root ports, drag from the component ports to the containing model boundary. When you release the connection, a root port is created at the boundary.

To manage Variant Component choices:

- By default, two variant choices are created when you create a Variant Component block. Right-click the Variant Component block and select **Variant > Label Mode Active Choice**, then select the active choice.
- To add an additional variant choice, right-click on the Variant Component block and select **Variant > Add Variant Choice**.
- Double-click into the Variant Component block to design the variants within it.
- Use the **Variant Manager** to easily switch between variant choices in a complex model hierarchy. Right-click on the Variant Component block and select **Variant > Open in Variant Manager**. For more information, see “Variant Manager for Simulink”.

## Ports

### Input

**Source** — Input connection from another component interface

If you connect to a source component, the interfaces on the ports are shared.

## Output

**Destination** — Output connection to another component interface

If you connect to a destination component, the interfaces on the ports are shared.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## Version History

Introduced in R2019a

## See Also

### Functions

`addVariantComponent` | `addChoice` | `getActiveChoice` | `getChoices` | `getCondition` | `setActiveChoice` | `setCondition` | `addPort` | `makeVariant` | `connect`

### Blocks

`Component` | `Reference Component` | `Adapter`

### Topics

“Decompose and Reuse Components”





# Objects

---

# systemcomposer.allocation.Allocation

Allocation between source element and target element

## Description

An Allocation object defines the allocation between the source element and the target element.

Related objects include:

- `systemcomposer.allocation.AllocationScenario`
- `systemcomposer.allocation.AllocationSet`

## Creation

Create two allocations between four elements in the default scenario, Scenario 1, using the allocate function.

```
defaultScenario = allocSet.getScenario("Scenario 1");  
defaultScenario.allocate(sourceElement1,sourceElement2);  
defaultScenario.allocate(sourceElement3,sourceElement4);
```

## Properties

### Source — Source element

element object

Source element, specified as a `systemcomposer.arch.Element` object.

### Target — Target element

element object

Target element, specified as a `systemcomposer.arch.Element` object.

### Scenario — Allocation scenario

allocation scenario object

Allocation scenario, specified as a `systemcomposer.allocation.AllocationScenario` object.

### UUID — Universal unique identifier

character vector

Universal unique identifier for allocation, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

## Object Functions

destroy Remove model element

## Examples

### Allocate Architectures in Tire Pressure Monitoring System

Use allocations to analyze a tire pressure monitoring system.

#### Overview

In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system must fulfill. We refer to this layer (or architecture) as the *functional architecture*. In this example, an automobile tire pressure monitoring system is described in three different architectures:

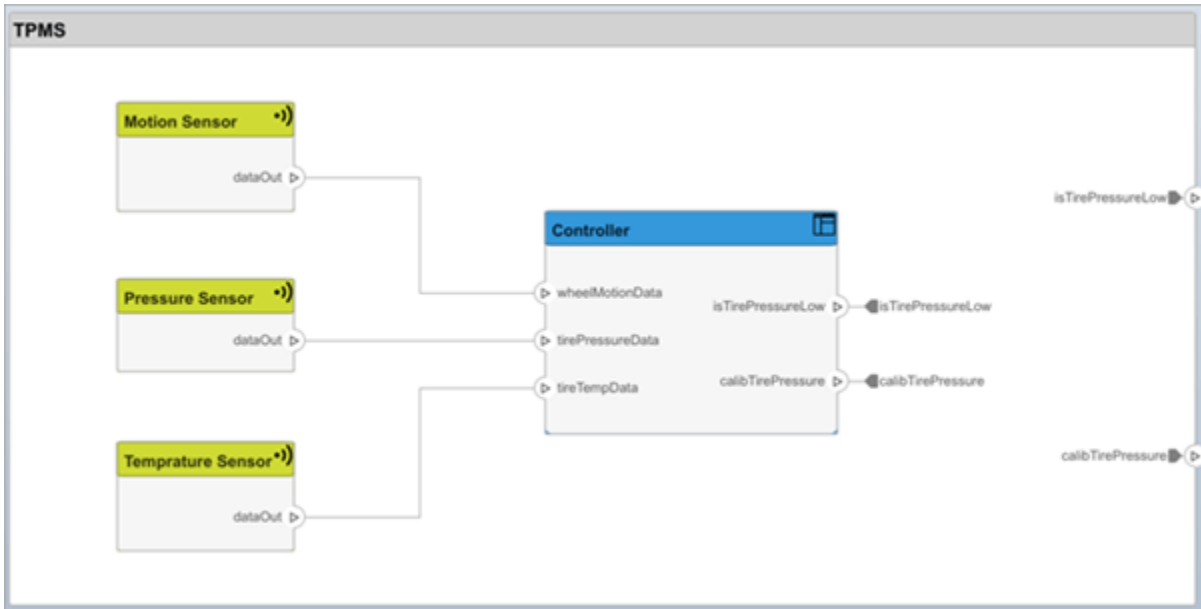
- 1** Functional Architecture — Describes the system in terms of its high-level functions. The connections show dependencies between functions [1].
- 2** Logical Architecture — Describes the system in terms of its logical components and how data is exchanged between them. Additionally, this architecture specifies behaviors for model simulation [2].
- 3** Platform Architecture — Describes the physical hardware needed for the system at a high level [3].

**Note:** This example illustrates allocations in System Composer™ using a specific methodology. However, you can use other methodologies that fit your needs.

The allocation process is defined as linking these three architectures that fully describe the system. The linking captures the information about each architectural layer and makes it accessible to the others.

Use this command to open the project.

```
scExampleTirePressureMonitorSystem
```



Open the FunctionalAllocation.mldatx file, which displays allocations from TPMS\_FunctionalArchitecture to TPMS\_LogicalArchitecture in the Allocation Editor. The elements of TPMS\_FunctionalArchitecture are displayed in the first column. The elements of TPMS\_LogicalArchitecture are displayed in the first row. The arrows indicate the allocations between model elements.

Scenario 1	TPMS_LogicalArchitec	TPMS Reporting S...	Right Front TPMS	Right Rear TPMS	Left Front TPMS	Left Rear TPMS	isTirePressureLow...2	calibTirePressure-->1	calibTirePressure-->1	isTirePressureLow...2	isTirePressureLow...2	calibTirePressure-->1	isTirePressureLow...2	calibTirePressure-->1
TPMS_FunctionalArchitecture														
Report Low Tire Pressure		↑												
InBus														
OutBus-->InBus														
OutBus-->InBus														
OutBus-->InBus														
Measure Tire Pressure			↑	↑	↑	↑								
Report Tire Pressure Levels		↑												
Calculate if pressure is low		↑												

The arrows display allocated components in the model. You can observe allocations for each element in the model hierarchy.

The rest of the example shows how to use this allocation information to further analyze the model.

## Functional to Logical Allocation and Coverage Analysis

This section shows how to perform coverage analysis to verify that all functions have been allocated. This process requires using the allocation information specified between the functional and logical architectures.

To start the analysis, load the allocation set.

```
allocSet = systemcomposer.allocation.load('FunctionalAllocation');
scenario = allocSet.Scenarios;
```

Verify that each function in the system is allocated.

```
import systemcomposer.query.*;
[~, allFunctions] = allocSet.SourceModel.find(HasStereotype(IsStereotypeDerivedFrom("TPMSProfile")));
unAllocatedFunctions = [];
for i = 1:numel(allFunctions)
    if isempty(scenario.getAllocatedTo(allFunctions(i)))
        unAllocatedFunctions = [unAllocatedFunctions allFunctions(i)];
    end
end

if isempty(unAllocatedFunctions)
    fprintf('All functions are allocated');
else
    fprintf('%d Functions have not been allocated', numel(unAllocatedFunctions));
end
```

All functions are allocated

The result displays All functions are allocated to verify that all functions in the system are allocated.

## Analyze Suppliers Providing Functions

This section shows how to identify which functions will be provided by which suppliers using the specified allocations. Since suppliers will be delivering these components to the system integrator, the supplier information is stored in the logical model.

```
suppliers = {'Supplier A', 'Supplier B', 'Supplier C', 'Supplier D'};
functionNames = arrayfun(@(x) x.Name, allFunctions, 'UniformOutput', false);
numFunNames = length(allFunctions);
numSuppliers = length(suppliers);
allocTable = table('Size', [numFunNames, numSuppliers], 'VariableTypes', repmat("double", 1, numFunNames));
allocTable.Properties.VariableNames = suppliers;
allocTable.Properties.RowNames = functionNames;
for i = 1:numFunNames
    elem = scenario.getAllocatedTo(allFunctions(i));
    for j = 1:numel(elem)
        elemSupplier = elem(j).getEvaluatedPropertyValue("TPMSProfile.LogicalComponent.Supplier");
        allocTable{i, strcmp(elemSupplier, suppliers)} = 1;
    end
end
```

The table shows which suppliers are responsible for the corresponding functions.

```
allocTable
```

```
allocTable=8×4 table
```

	Supplier A	Supplier B	Supplier C	Supplier D
Measure temprature of tire	0	0	0	1
Measure pressure on tire	0	0	1	0
Calculate Tire Pressure	0	1	0	0
Report Tire Pressure Levels	1	0	0	0
Measure rotations	0	1	0	0
Measure Tire Pressure	0	0	0	0
Report Low Tire Pressure	1	0	0	0
Calculate if pressure is low	1	0	0	0

### Analyze Software Deployment Strategies

You can determine if the Engine Control Unit (ECU) has enough capacity to house all the software components. The software components are allocated to the cores themselves, but the ECU is the component that has the budget property.

Get the platform architecture.

```
platformArch = systemcomposer.loadModel('PlatformArchitecture');
```

Load the allocation.

```
softwareDeployment = systemcomposer.allocation.load('SoftwareDeployment');

frontECU = platformArch.lookup('Path', 'PlatformArchitecture/Front ECU');
rearECU = platformArch.lookup('Path', 'PlatformArchitecture/Rear ECU');

scenario1 = softwareDeployment.getScenario('Scenario 1');
scenario2 = softwareDeployment.getScenario('Scenario 2');
frontECU_availMemory = frontECU.getEvaluatedPropertyValue("TPMSProfile.ECU.MemoryCapacity");
rearECU_availMemory = rearECU.getEvaluatedPropertyValue("TPMSProfile.ECU.MemoryCapacity");

frontECU_memoryUsed1 = getUtilizedMemoryOnECU(frontECU, scenario1);
frontECU_isOverBudget1 = frontECU_memoryUsed1 > frontECU_availMemory;
rearECU_memoryUsed1 = getUtilizedMemoryOnECU(rearECU, scenario1);
rearECU_isOverBudget1 = rearECU_memoryUsed1 > rearECU_availMemory;

frontECU_memoryUsed2 = getUtilizedMemoryOnECU(frontECU, scenario2);
frontECU_isOverBudget2 = frontECU_memoryUsed2 > frontECU_availMemory;
rearECU_memoryUsed2 = getUtilizedMemoryOnECU(rearECU, scenario2);
rearECU_isOverBudget2 = rearECU_memoryUsed2 > rearECU_availMemory;
```

Build a table to showcase the results.

```
softwareDeploymentTable = table([frontECU_memoryUsed1;frontECU_availMemory; ...
    frontECU_isOverBudget1;rearECU_memoryUsed1;rearECU_availMemory;rearECU_isOverBudget1], ...
    [frontECU_memoryUsed2; frontECU_availMemory; frontECU_isOverBudget2;rearECU_memoryUsed2; .
    rearECU_availMemory; rearECU_isOverBudget2], ...
    'VariableNames',{ 'Scenario 1','Scenario 2'},...
    'RowNames', {'Front ECU Memory Used (MB)', 'Front ECU Memory (MB)', 'Front ECU Overloaded',
    'Rear ECU Memory Used (MB)', 'Rear ECU Memory (MB)', 'Rear ECU Overloaded'})

softwareDeploymentTable=6×2 table
                Scenario 1    Scenario 2
```

Front ECUMemory Used (MB)	110	90
Front ECU Memory (MB)	100	100
Front ECU Overloaded	1	0
Rear ECU Memory Used (MB)	0	20
Rear ECU Memory (MB)	100	100
Rear ECU Overloaded	0	0

```
function memoryUsed = getUtilizedMemoryOnECU(ecu, scenario)
```

For each component in the ECU, accumulate the binary size required for each allocated software component.

```
coreNames = {'Core1', 'Core2', 'Core3', 'Core4'};
memoryUsed = 0;
for i = 1:numel(coreNames)
    core = ecu.Model.lookup('Path', [ecu.getQualifiedName '/' coreNames{i}]);
    allocatedSWComps = scenario.getAllocatedFrom(core);
    for j = 1:numel(allocatedSWComps)
        binarySize = allocatedSWComps(j).getEvaluatedPropertyValue("TPMSProfile.SWComponent.BinarySize");
        memoryUsed = memoryUsed + binarySize;
    end
end
end
```

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## **Version History**

**Introduced in R2020b**

### **See Also**

`getAllocatedFrom` | `getAllocation` | `getAllocatedTo` | `allocate` | `getScenario`

### **Topics**

“Create and Manage Allocations Programmatically”



# systemcomposer.allocation.AllocationScenario

Allocation scenario

## Description

An `AllocationScenario` object defines a collection of allocations between elements in the source model to elements in the target model.

## Creation

Create an allocation set with name `myNewAllocation` using the `systemcomposer.allocation.createAllocationSet` function.

```
systemcomposer.allocation.createAllocationSet("myNewAllocation", ...
    "Source_Model_Allocation", "Target_Model_Allocation");
```

Create a second allocation scenario `Scenario 2` in addition to the default scenario `Scenario 1` using the `createScenario` function.

```
scenario = createScenario(myAllocationSet, "Scenario 2")
```

## Properties

### Name — Name of allocation scenario

character vector

Name of allocation scenario, specified as a character vector.

Example: 'Scenario 1'

Data Types: char

### Allocations — Allocations in scenario

array of allocation objects

Allocations in scenario, specified as an array of `systemcomposer.allocation.Allocation` objects.

### AllocationSet — Allocation set to which scenario belongs

allocation set object

Allocation set to which scenario belongs, specified as an `systemcomposer.allocation.AllocationSet` object.

### Description — Description of allocation scenario

character vector

Description of allocation scenario, specified as a character vector.

Data Types: char

### UUID — Universal unique identifier

character vector

Universal unique identifier for allocation scenario, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### Object Functions

allocate	Create new allocation
deallocate	Delete allocation
getAllocation	Get allocation between source and target elements
getAllocatedFrom	Get allocation source
getAllocatedTo	Get allocation target
destroy	Remove model element

### Examples

#### Allocate Architectures in Tire Pressure Monitoring System

Use allocations to analyze a tire pressure monitoring system.

#### Overview

In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system must fulfill. We refer to this layer (or architecture) as the *functional architecture*. In this example, an automobile tire pressure monitoring system is described in three different architectures:

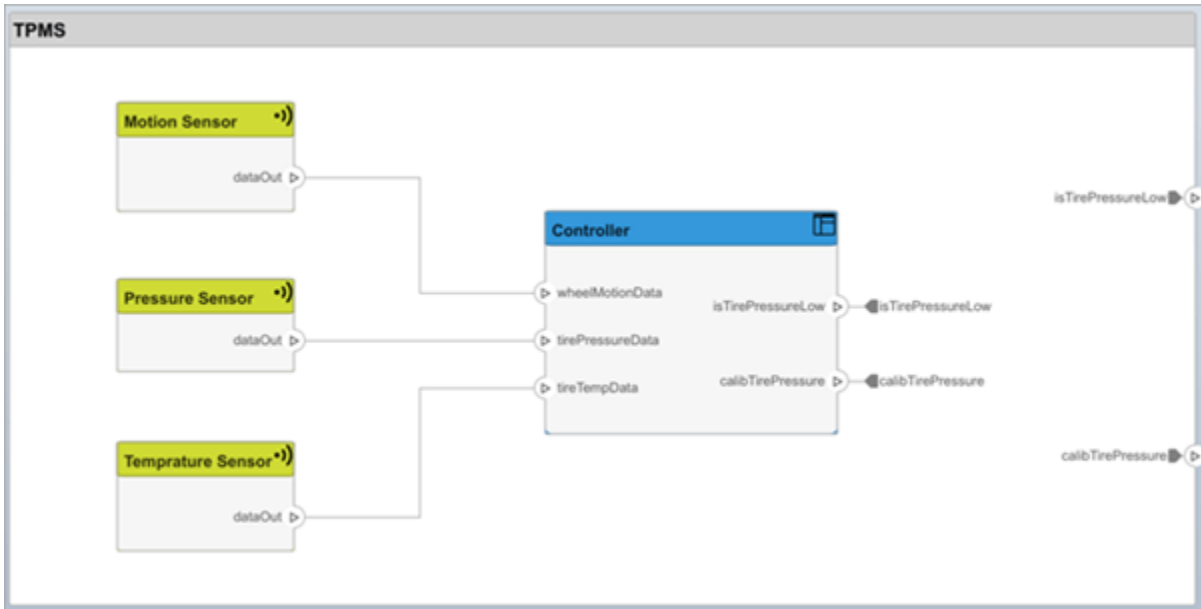
- 1 Functional Architecture — Describes the system in terms of its high-level functions. The connections show dependencies between functions [1].
- 2 Logical Architecture — Describes the system in terms of its logical components and how data is exchanged between them. Additionally, this architecture specifies behaviors for model simulation [2].
- 3 Platform Architecture — Describes the physical hardware needed for the system at a high level [3].

**Note:** This example illustrates allocations in System Composer™ using a specific methodology. However, you can use other methodologies that fit your needs.

The allocation process is defined as linking these three architectures that fully describe the system. The linking captures the information about each architectural layer and makes it accessible to the others.

Use this command to open the project.

```
scExampleTirePressureMonitorSystem
```



Open the FunctionalAllocation.mldatx file, which displays allocations from TPMS\_FunctionalArchitecture to TPMS\_LogicalArchitecture in the Allocation Editor. The elements of TPMS\_FunctionalArchitecture are displayed in the first column. The elements of TPMS\_LogicalArchitecture are displayed in the first row. The arrows indicate the allocations between model elements.

Scenario 1	TPMS_LogicalArchitec	TPMS Reporting S	Right Front TPMS	Right Rear TPMS	Left Front TPMS	Left Rear TPMS	isTirePressureLow-->	calibTirePressure-->1	calibTirePressure-->1	isTirePressureLow-->	isTirePressureLow-->	calibTirePressure-->1	isTirePressureLow-->	calibTirePressure-->1
TPMS_FunctionalArchitecture														
Report Low Tire Pressure		↑												
InBus														
OutBus-->InBus														
OutBus-->InBus														
OutBus-->InBus														
Measure Tire Pressure			↑	↑	↑	↑								
Report Tire Pressure Levels		↑												
Calculate If pressure is low		↑												

The arrows display allocated components in the model. You can observe allocations for each element in the model hierarchy.

The rest of the example shows how to use this allocation information to further analyze the model.

## Functional to Logical Allocation and Coverage Analysis

This section shows how to perform coverage analysis to verify that all functions have been allocated. This process requires using the allocation information specified between the functional and logical architectures.

To start the analysis, load the allocation set.

```
allocSet = systemcomposer.allocation.load('FunctionalAllocation');
scenario = allocSet.Scenarios;
```

Verify that each function in the system is allocated.

```
import systemcomposer.query.*;
[~, allFunctions] = allocSet.SourceModel.find(HasStereotype(IsStereotypeDerivedFrom("TPMSProfile")));
unAllocatedFunctions = [];
for i = 1:numel(allFunctions)
    if isempty(scenario.getAllocatedTo(allFunctions(i)))
        unAllocatedFunctions = [unAllocatedFunctions allFunctions(i)];
    end
end

if isempty(unAllocatedFunctions)
    fprintf('All functions are allocated');
else
    fprintf('%d Functions have not been allocated', numel(unAllocatedFunctions));
end
```

All functions are allocated

The result displays All functions are allocated to verify that all functions in the system are allocated.

## Analyze Suppliers Providing Functions

This section shows how to identify which functions will be provided by which suppliers using the specified allocations. Since suppliers will be delivering these components to the system integrator, the supplier information is stored in the logical model.

```
suppliers = {'Supplier A', 'Supplier B', 'Supplier C', 'Supplier D'};
functionNames = arrayfun(@(x) x.Name, allFunctions, 'UniformOutput', false);
numFunNames = length(allFunctions);
numSuppliers = length(suppliers);
allocTable = table('Size', [numFunNames, numSuppliers], 'VariableTypes', repmat("double", 1, numFunNames));
allocTable.Properties.VariableNames = suppliers;
allocTable.Properties.RowNames = functionNames;
for i = 1:numFunNames
    elem = scenario.getAllocatedTo(allFunctions(i));
    for j = 1:numel(elem)
        elemSupplier = elem(j).getEvaluatedPropertyValue("TPMSProfile.LogicalComponent.Supplier");
        allocTable{i, strcmp(elemSupplier, suppliers)} = 1;
    end
end
```

The table shows which suppliers are responsible for the corresponding functions.

```
allocTable
```

```
allocTable=8×4 table
```

	Supplier A	Supplier B	Supplier C	Supplier D
Measure temprature of tire	0	0	0	1
Measure pressure on tire	0	0	1	0
Calculate Tire Pressure	0	1	0	0
Report Tire Pressure Levels	1	0	0	0
Measure rotations	0	1	0	0
Measure Tire Pressure	0	0	0	0
Report Low Tire Pressure	1	0	0	0
Calculate if pressure is low	1	0	0	0

## Analyze Software Deployment Strategies

You can determine if the Engine Control Unit (ECU) has enough capacity to house all the software components. The software components are allocated to the cores themselves, but the ECU is the component that has the budget property.

Get the platform architecture.

```
platformArch = systemcomposer.loadModel('PlatformArchitecture');
```

Load the allocation.

```
softwareDeployment = systemcomposer.allocation.load('SoftwareDeployment');

frontECU = platformArch.lookup('Path', 'PlatformArchitecture/Front ECU');
rearECU = platformArch.lookup('Path', 'PlatformArchitecture/Rear ECU');

scenario1 = softwareDeployment.getScenario('Scenario 1');
scenario2 = softwareDeployment.getScenario('Scenario 2');
frontECU_availMemory = frontECU.getEvaluatedPropertyValue("TPMSProfile.ECU.MemoryCapacity");
rearECU_availMemory = rearECU.getEvaluatedPropertyValue("TPMSProfile.ECU.MemoryCapacity");

frontECU_memoryUsed1 = getUtilizedMemoryOnECU(frontECU, scenario1);
frontECU_isOverBudget1 = frontECU_memoryUsed1 > frontECU_availMemory;
rearECU_memoryUsed1 = getUtilizedMemoryOnECU(rearECU, scenario1);
rearECU_isOverBudget1 = rearECU_memoryUsed1 > rearECU_availMemory;

frontECU_memoryUsed2 = getUtilizedMemoryOnECU(frontECU, scenario2);
frontECU_isOverBudget2 = frontECU_memoryUsed2 > frontECU_availMemory;
rearECU_memoryUsed2 = getUtilizedMemoryOnECU(rearECU, scenario2);
rearECU_isOverBudget2 = rearECU_memoryUsed2 > rearECU_availMemory;
```

Build a table to showcase the results.

```
softwareDeploymentTable = table([frontECU_memoryUsed1;frontECU_availMemory; ...
    frontECU_isOverBudget1;rearECU_memoryUsed1;rearECU_availMemory;rearECU_isOverBudget1], ...
    [frontECU_memoryUsed2; frontECU_availMemory; frontECU_isOverBudget2;rearECU_memoryUsed2; .
    rearECU_availMemory; rearECU_isOverBudget2], ...
    'VariableNames',{ 'Scenario 1','Scenario 2'},...
    'RowNames', {'Front ECUMemory Used (MB)', 'Front ECU Memory (MB)', 'Front ECU Overloaded',
    'Rear ECU Memory Used (MB)', 'Rear ECU Memory (MB)', 'Rear ECU Overloaded'})
```

```
softwareDeploymentTable=6×2 table
```

```
Scenario 1 Scenario 2
```

Front ECUMemory Used (MB)	110	90
Front ECU Memory (MB)	100	100
Front ECU Overloaded	1	0
Rear ECU Memory Used (MB)	0	20
Rear ECU Memory (MB)	100	100
Rear ECU Overloaded	0	0

```
function memoryUsed = getUtilizedMemoryOnECU(ecu, scenario)
```

For each component in the ECU, accumulate the binary size required for each allocated software component.

```
coreNames = {'Core1', 'Core2', 'Core3', 'Core4'};
memoryUsed = 0;
for i = 1:numel(coreNames)
    core = ecu.Model.lookup('Path', [ecu.getQualifiedName '/' coreNames{i}]);
    allocatedSWComps = scenario.getAllocatedFrom(core);
    for j = 1:numel(allocatedSWComps)
        binarySize = allocatedSWComps(j).getEvaluatedPropertyValue("TPMSProfile.SWComponent.BinarySize");
        memoryUsed = memoryUsed + binarySize;
    end
end
end
```

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## **Version History**

**Introduced in R2020b**

### **See Also**

`createScenario`

### **Topics**

“Create and Manage Allocations Programmatically”

# systemcomposer.allocation.AllocationSet

Set of allocation scenarios

## Description

An AllocationSet object defines a collection of allocation scenarios between two System Composer models.

## Creation

Create an allocation set with name myNewAllocation using the systemcomposer.allocation.createAllocationSet function.

```
systemcomposer.allocation.createAllocationSet("myNewAllocation", ...  
    "Source_Model_Allocation", "Target_Model_Allocation");
```

## Properties

### Name — Name of allocation set

character vector

Name of allocation set, specified as a character vector.

Example: 'MyNewAllocation'

Data Types: char

### SourceModel — Source model for allocation

model object

Source model for allocation, specified as a systemcomposer.arch.Model object.

### TargetModel — Target model for allocation

model object

Target model for allocation, specified as a systemcomposer.arch.Model object.

### Scenarios — Allocation scenarios

array of allocation scenario objects

Allocation scenarios, specified as an array of systemcomposer.allocation.AllocationScenario objects.

### Description — Description of allocation set

character vector

Description of allocation set, specified as a character vector.

Data Types: char



**NeedsRefresh — Whether allocation set is out of date**`true or 1 | false or 0`

Whether allocation set is out of date with the source model, target model, or both, specified as a logical.

Data Types: `logical`

**Dirty — Whether allocation has unsaved changes**`true or 1 | false or 0`

Whether allocation set has unsaved changes, specified as a logical.

Data Types: `logical`

**UUID — Universal unique identifier**`character vector`

Universal unique identifier for allocation set, specified as a character vector.

Example: `'91d5de2c-b14c-4c76-a5d6-5dd0037c52df'`

Data Types: `char`

**Object Functions**

<code>createScenario</code>	Create new empty allocation scenario
<code>getScenario</code>	Get allocation scenario
<code>deleteScenario</code>	Delete allocation scenario
<code>synchronizeChanges</code>	Synchronize changes of models in allocation set
<code>find</code>	Find loaded allocation set
<code>save</code>	Save allocation set as file
<code>close</code>	Close allocation set
<code>closeAll</code>	Close all open allocation sets

**Examples****Allocate Architectures in Tire Pressure Monitoring System**

Use allocations to analyze a tire pressure monitoring system.

**Overview**

In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system must fulfill. We refer to this layer (or architecture) as the *functional architecture*. In this example, an automobile tire pressure monitoring system is described in three different architectures:

- 1 Functional Architecture — Describes the system in terms of its high-level functions. The connections show dependencies between functions [1].
- 2 Logical Architecture — Describes the system in terms of its logical components and how data is exchanged between them. Additionally, this architecture specifies behaviors for model simulation [2].

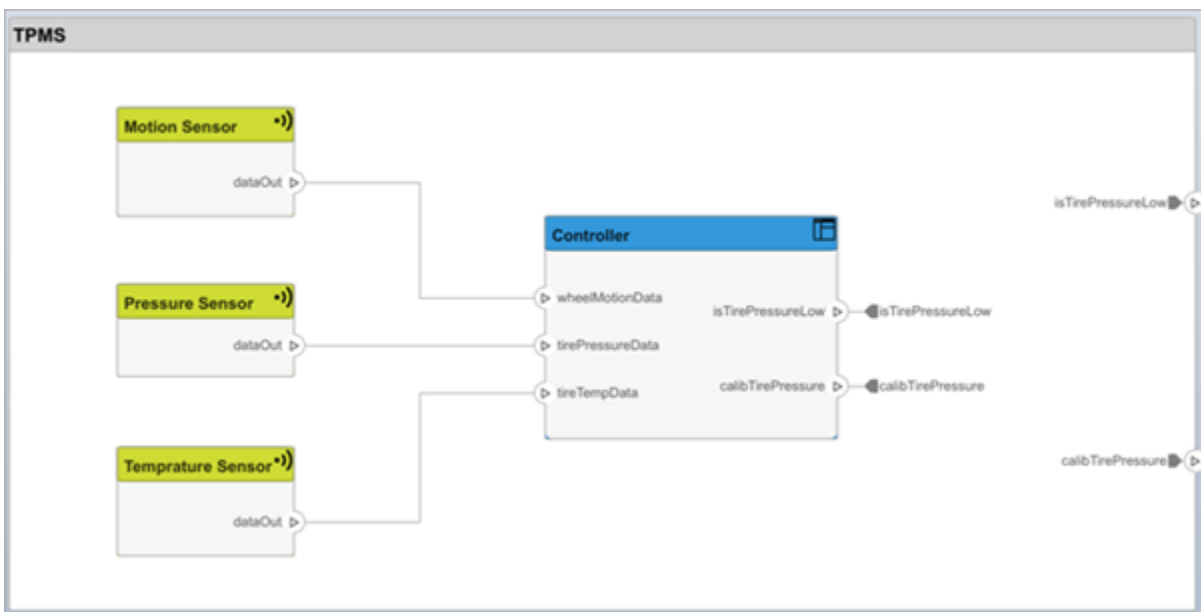
- 3 Platform Architecture — Describes the physical hardware needed for the system at a high level [3].

**Note:** This example illustrates allocations in System Composer™ using a specific methodology. However, you can use other methodologies that fit your needs.

The allocation process is defined as linking these three architectures that fully describe the system. The linking captures the information about each architectural layer and makes it accessible to the others.

Use this command to open the project.

```
scExampleTirePressureMonitorSystem
```



Open the `FunctionalAllocation.mldatx` file, which displays allocations from `TPMS_FunctionalArchitecture` to `TPMS_LogicalArchitecture` in the Allocation Editor. The elements of `TPMS_FunctionalArchitecture` are displayed in the first column. The elements of `TPMS_LogicalArchitecture` are displayed in the first row. The arrows indicate the allocations between model elements.

Scenario 1															
		TPMS_LogicalArchitec													
		TPMS Reporting S													
		Right Front TPMS													
		Right Rear TPMS													
		Left Front TPMS													
		Left Rear TPMS													
		isTirePressureLow->													
		calibTirePressure->1													
		calibTirePressure->1													
		isTirePressureLow->													
		isTirePressureLow->													
		calibTirePressure->1													
		calibTirePressure->1													
		TPMS_FunctionalArchitecture													
		Report Low Tire Pressure													
		InBus													
		OutBus->InBus													
		OutBus->InBus													
		OutBus->InBus													
		Measure Tire Pressure													
		Report Tire Pressure Levels													
		Calculate if pressure is low													

The arrows display allocated components in the model. You can observe allocations for each element in the model hierarchy.

The rest of the example shows how to use this allocation information to further analyze the model.

### Functional to Logical Allocation and Coverage Analysis

This section shows how to perform coverage analysis to verify that all functions have been allocated. This process requires using the allocation information specified between the functional and logical architectures.

To start the analysis, load the allocation set.

```
allocSet = systemcomposer.allocation.load('FunctionalAllocation');
scenario = allocSet.Scenarios;
```

Verify that each function in the system is allocated.

```
import systemcomposer.query.*;
[~, allFunctions] = allocSet.SourceModel.find(HasStereotype(IsStereotypeDerivedFrom("TPMSProfi
unAllocatedFunctions = [];
for i = 1:numel(allFunctions)
    if isempty(scenario.getAllocatedTo(allFunctions(i)))
        unAllocatedFunctions = [unAllocatedFunctions allFunctions(i)];
    end
end

if isempty(unAllocatedFunctions)
    fprintf('All functions are allocated');
else
    fprintf('%d Functions have not been allocated', numel(unAllocatedFunctions));
end
```

All functions are allocated

The result displays All functions are allocated to verify that all functions in the system are allocated.

### Analyze Suppliers Providing Functions

This section shows how to identify which functions will be provided by which suppliers using the specified allocations. Since suppliers will be delivering these components to the system integrator, the supplier information is stored in the logical model.

```
suppliers = {'Supplier A', 'Supplier B', 'Supplier C', 'Supplier D'};
functionNames = arrayfun(@(x) x.Name, allFunctions, 'UniformOutput', false);
numFunNames = length(allFunctions);
numSuppliers = length(suppliers);
allocTable = table('Size', [numFunNames, numSuppliers], 'VariableTypes', repmat("double", 1, numFunNames + numSuppliers));
allocTable.Properties.VariableNames = suppliers;
allocTable.Properties.RowNames = functionNames;
for i = 1:numFunNames
    elem = scenario.getAllocatedTo(allFunctions(i));
    for j = 1: numel(elem)
        elemSupplier = elem(j).getEvaluatedPropertyValue("TPMSProfile.LogicalComponent.Supplier");
        allocTable{i, strcmp(elemSupplier, suppliers)} = 1;
    end
end
```

The table shows which suppliers are responsible for the corresponding functions.

```
allocTable
```

allocTable=8×4 table

	Supplier A	Supplier B	Supplier C	Supplier D
Measure temprature of tire	0	0	0	1
Measure pressure on tire	0	0	1	0
Calculate Tire Pressure	0	1	0	0
Report Tire Pressure Levels	1	0	0	0
Measure rotations	0	1	0	0
Measure Tire Pressure	0	0	0	0
Report Low Tire Pressure	1	0	0	0
Calculate if pressure is low	1	0	0	0

### Analyze Software Deployment Strategies

You can determine if the Engine Control Unit (ECU) has enough capacity to house all the software components. The software components are allocated to the cores themselves, but the ECU is the component that has the budget property.

Get the platform architecture.

```
platformArch = systemcomposer.loadModel('PlatformArchitecture');
```

Load the allocation.

```
softwareDeployment = systemcomposer.allocation.load('SoftwareDeployment');
frontECU = platformArch.lookup('Path', 'PlatformArchitecture/Front ECU');
```

```

rearECU = platformArch.lookup('Path', 'PlatformArchitecture/Rear ECU');

scenario1 = softwareDeployment.getScenario('Scenario 1');
scenario2 = softwareDeployment.getScenario('Scenario 2');
frontECU_availMemory = frontECU.getEvaluatedPropertyValue("TPMSProfile.ECU.MemoryCapacity");
rearECU_availMemory = rearECU.getEvaluatedPropertyValue("TPMSProfile.ECU.MemoryCapacity");

frontECU_memoryUsed1 = getUtilizedMemoryOnECU(frontECU, scenario1);
frontECU_isOverBudget1 = frontECU_memoryUsed1 > frontECU_availMemory;
rearECU_memoryUsed1 = getUtilizedMemoryOnECU(rearECU, scenario1);
rearECU_isOverBudget1 = rearECU_memoryUsed1 > rearECU_availMemory;

frontECU_memoryUsed2 = getUtilizedMemoryOnECU(frontECU, scenario2);
frontECU_isOverBudget2 = frontECU_memoryUsed2 > frontECU_availMemory;
rearECU_memoryUsed2 = getUtilizedMemoryOnECU(rearECU, scenario2);
rearECU_isOverBudget2 = rearECU_memoryUsed2 > rearECU_availMemory;

```

Build a table to showcase the results.

```

softwareDeploymentTable = table([frontECU_memoryUsed1;frontECU_availMemory; ...
    frontECU_isOverBudget1;rearECU_memoryUsed1;rearECU_availMemory;rearECU_isOverBudget1], ...
    [frontECU_memoryUsed2; frontECU_availMemory; frontECU_isOverBudget2;rearECU_memoryUsed2; ...
    rearECU_availMemory; rearECU_isOverBudget2], ...
    'VariableNames',{'Scenario 1','Scenario 2'},...
    'RowNames', {'Front ECUMemory Used (MB)', 'Front ECU Memory (MB)', 'Front ECU Overloaded',
    'Rear ECU Memory Used (MB)', 'Rear ECU Memory (MB)', 'Rear ECU Overloaded'})

```

softwareDeploymentTable=6x2 table

	Scenario 1	Scenario 2
	-----	-----
Front ECUMemory Used (MB)	110	90
Front ECU Memory (MB)	100	100
Front ECU Overloaded	1	0
Rear ECU Memory Used (MB)	0	20
Rear ECU Memory (MB)	100	100
Rear ECU Overloaded	0	0

```
function memoryUsed = getUtilizedMemoryOnECU(ecu, scenario)
```

For each component in the ECU, accumulate the binary size required for each allocated software component.

```

coreNames = {'Core1','Core2','Core3','Core4'};
memoryUsed = 0;
for i = 1:numel(coreNames)
    core = ecu.Model.lookup('Path', [ecu.getQualifiedName '/' coreNames{i}]);
    allocatedSWComps = scenario.getAllocatedFrom(core);
    for j = 1:numel(allocatedSWComps)
        binarySize = allocatedSWComps(j).getEvaluatedPropertyValue("TPMSProfile.SWComponent.BinarySize");
        memoryUsed = memoryUsed + binarySize;
    end
end

```

end

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

systemcomposer.allocation.Allocation |  
systemcomposer.allocation.AllocationScenario | editor | createAllocationSet

### Topics

“Create and Manage Allocations Programmatically”

# systemcomposer.analysis.ArchitectureInstance

Architecture in analysis instance

## Description

An ArchitectureInstance object represents an instance of an architecture.

## Creation

Create an instance of an architecture using the `instantiate` function.

```
instance = instantiate(model.Architecture, 'LatencyProfile', 'NewInstance', ...  
    'Function', @calculateLatency, 'Arguments', '3', 'Strict', true, ...  
    'NormalizeUnits', false, 'Direction', 'PreOrder')
```

## Properties

### Name — Name of instance

character vector

Name of instance, specified as a character vector.

Example: 'NewInstance'

Data Types: char

### Components — Child components of instance

array of component instance objects

Child components of instance, specified as an array of `systemcomposer.analysis.ComponentInstance` objects.

### Ports — Ports of architecture instance

array of port instance objects

Ports of architecture instance, specified as an array of `systemcomposer.analysis.PortInstance` objects.

### Connectors — Connectors in architecture instance

array of connector instance objects

Connectors in architecture instance, specified as an array of `systemcomposer.analysis.ConnectorInstance` objects, connecting child components.

### Specification — Reference to architecture in design model

architecture object

Reference to architecture in design model, specified as a `systemcomposer.arch.Architecture` object.

**NormalizeUnits — Whether units normalize**

true or 1 | false or 0

Whether units normalize the value of properties in the instantiation, specified as a logical.

Data Types: logical

**IsStrict — Whether instances get properties**

true or 1 | false or 0

Whether instances get properties if the specification of the instance has the stereotype applied, specified as a logical.

Data Types: logical

**AnalysisFunction — Analysis function**

MATLAB® function handle

Analysis function, specified as the MATLAB function handle to be executed when analysis is run.

Example: @calculateLatency

**AnalysisDirection — Analysis direction**

enumeration | character vector

Analysis direction, specified as one of the following enumerations:

- systemcomposer.IteratorDirection.TopDown
- systemcomposer.IteratorDirection.BottomUp
- systemcomposer.IteratorDirection.PreOrder
- systemcomposer.IteratorDirection.PostOrder

or a character vector of one of the following options: 'TopDown', 'PreOrder', 'PostOrder', or 'BottomUp'

Data Types: enum | char

**AnalysisArguments — Analysis arguments**

character vector

Analysis arguments, specified as a character vector of optional arguments to the analysis function.

Example: '3'

Data Types: char

**ImmediateUpdate — Whether analysis instance updates automatically**

true or 1 | false or 0

Whether analysis viewer updates automatically when the design model changes, specified as a logical.

Data Types: logical

**Object Functions**

getValue            Get value of property from element instance



setValue	Set value of property for element instance
hasValue	Find if element instance has property value
iterate	Iterate over model elements
lookup	Search for architectural element
save	Save architecture instance
update	Update architecture model
refresh	Refresh architecture instance
isArchitecture	Find if instance is architecture instance
isComponent	Find if instance is component instance
isConnector	Find if instance is connector instance
isPort	Find if instance is port instance

## Examples

### Analyze Latency Characteristics

Create an instantiation for analysis for a system with latency in its wiring. The materials used are copper, fiber, and WiFi.

### Create Latency Profile with Stereotypes and Properties

Create a System Composer profile with a base, connector, component, and port stereotype. Add properties with default values to each stereotype as needed for analysis.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfileC");
```

Add a base stereotype with properties.

```
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");
```

Add a connector stereotype with properties.

```
connLatency = profile.addStereotype("ConnectorLatency",...
    Parent="LatencyProfileC.LatencyBase");
connLatency.addProperty("secure",Type="boolean",DefaultValue="true");
connLatency.addProperty("linkDistance",Type="double");
```

Add a component stereotype with properties.

```
nodeLatency = profile.addStereotype("NodeLatency",...
    Parent="LatencyProfileC.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");
```

Add a port stereotype with properties.

```
portLatency = profile.addStereotype("PortLatency",...
    Parent="LatencyProfileC.LatencyBase");
portLatency.addProperty("queueDepth",Type="double",DefaultValue="4.29");
portLatency.addProperty("dummy",Type="int32");
```

### Instantiate Using Analysis Function

Create a new model and apply the profile. Create components, ports, and connections in the model. Apply stereotypes to the model elements. Finally, instantiate using the analysis function.

```
model = systemcomposer.createModel("archModel",true);
arch = model.Architecture;
```

Apply profile to model.

```
model.applyProfile("LatencyProfileC");
```

Create components, ports, and connections.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},{'in','out'});

componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower','MotionCommand'});
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},{'in','out'});

c_sensorData = connect(arch,componentSensor,componentPlanning);
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

Clean up the canvas.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

Batch apply stereotypes to model elements.

```
batchApplyStereotype(arch,"Component","LatencyProfileC.NodeLatency");
batchApplyStereotype(arch,"Port","LatencyProfileC.PortLatency");
batchApplyStereotype(arch,"Connector","LatencyProfileC.ConnectorLatency");
```

Instantiate using the analysis function.

```
instance = instantiate(model.Architecture,"LatencyProfileC","NewInstance",...
    Function=@calculateLatency,Arguments="3", ...
    Strict=true,NormalizeUnits=false,Direction="PreOrder")
```

```
instance =
```

```
ArchitectureInstance with properties:
```

```
    Specification: [1x1 systemcomposer.arch.Architecture]
        IsStrict: 1
    NormalizeUnits: 0
    AnalysisFunction: @calculateLatency
    AnalysisDirection: PreOrder
    AnalysisArguments: '3'
    ImmediateUpdate: 0
        Components: [1x3 systemcomposer.analysis.ComponentInstance]
            Ports: [0x0 systemcomposer.analysis.PortInstance]
        Connectors: [1x3 systemcomposer.analysis.ConnectorInstance]
            Name: 'NewInstance'
```

### Inspect Component, Port, and Connector Instances

Get properties from component, port, and connector instances.

```
defaultResources = instance.Components(1).getValue("LatencyProfileC.NodeLatency.resources")
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue("LatencyProfileC.ConnectorLatency.secure")
defaultSecure = logical
1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue("LatencyProfileC.PortLatency.queueDepth")
defaultQueueDepth = 4.2900
```

## Battery Sizing and Automotive Electrical System Analysis

### Overview

Model a typical automotive electrical system as an architectural model and run a primitive analysis. The elements in the model can be broadly grouped as either a source or a load. Various properties of the sources and loads are set as part of the stereotype. This example uses the `iterate` method of the specification API to iterate through each element of the model and run analysis using the stereotype properties.

### Structure of Model

The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set on each component, the following analyses are performed:

- Total KeyOffLoad.
- Number of days required for KeyOffLoad to discharge 30% of the battery.
- Total CrankingInRush current.
- Total Cranking current.
- Ability of the battery to start the vehicle at 0°F based on the battery cold cranking amps (CCA). The discharge time is computed based on Puekert coefficient (k), which describes the relationship between the rate of discharge and the available capacity of the battery.

### Load Model and Run Analysis

```
scExampleAutomotiveElectricalSystemAnalysis
archModel = systemcomposer.loadModel('scExampleAutomotiveElectricalSystemAnalysis');
```

Instantiate battery sizing class used by the analysis function to store analysis results.

```
objcomputeBatterySizing = computeBatterySizing;
```

Run the analysis using the iterator.

```
archModel.iterate('Topdown',@computeLoad,objcomputeBatterySizing)
```

Display analysis results.

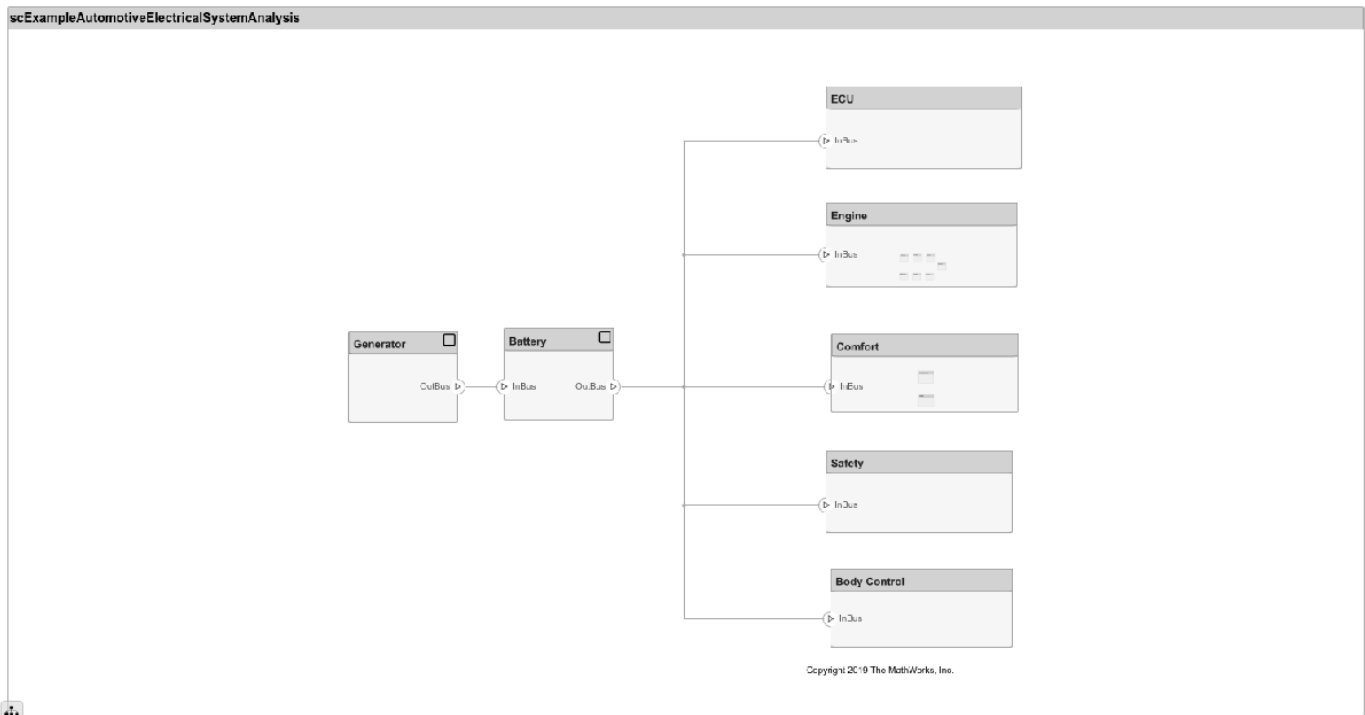
```
objcomputeBatterySizing.displayResults
```

```
Total KeyOffLoad: 158.708 mA
Number of days required for KeyOffLoad to discharge 30% of battery: 55.789.
```

Total CrankingInRush current: 70 A  
 Total Cranking current: 104 A  
 CCA of the specified battery is sufficient to start the car at 0 F.

```
ans =
  computeBatterySizing with properties:

    totalCrankingInrushCurrent: 70
    totalCrankingCurrent: 104
    totalAccesoriesCurrent: 71.6667
    totalKeyOffLoad: 158.7080
    batteryCCA: 500
    batteryCapacity: 850
    puekertcoefficient: 1.2000
```



**Close Model**

```
bdclose('scExampleAutomotiveElectricalSystemAnalysis');
```

**More About****Definitions**

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

deleteInstance | instantiate | loadInstance |  
systemcomposer.analysis.ComponentInstance |  
systemcomposer.analysis.PortInstance |  
systemcomposer.analysis.ConnectorInstance | systemcomposer.analysis.Instance

### Topics

"Write Analysis Function"

# systemcomposer.analysis.ComponentInstance

Component in analysis instance

## Description

A ComponentInstance object represents an instance of a component.

## Creation

Create an instance of an architecture using the `instantiate` function.

```
instance = instantiate(model.Architecture,'LatencyProfile','NewInstance', ...  
'Function',@calculateLatency,'Arguments','3','Strict',true, ...  
'NormalizeUnits',false,'Direction','PreOrder')
```

## Properties

### Name — Name of instance

character vector

Name of instance, specified as a character vector.

Example: 'NewInstance'

Data Types: char

### Components — Child components of instance

array of component instance objects

Child components of instance, specified as an array of `systemcomposer.analysis.ComponentInstance` objects.

### Ports — Ports of component instance

array of port instance objects

Ports of component instance, specified as an array of `systemcomposer.analysis.PortInstance` objects.

### Connectors — Connectors in component instance

array of connector instance objects

Connectors in component instance that connect child components, specified as an array of `systemcomposer.analysis.ConnectorInstance` objects.

### Parent — Parent of component

architecture instance object

Parent of component, specified as a `systemcomposer.analysis.ArchitectureInstance` object.

### Specification — Reference to component in design model

component object



Reference to component in design model, specified as a `systemcomposer.arch.Component` object.

## Object Functions

<code>getValue</code>	Get value of property from element instance
<code>setValue</code>	Set value of property for element instance
<code>hasValue</code>	Find if element instance has property value
<code>isArchitecture</code>	Find if instance is architecture instance
<code>isComponent</code>	Find if instance is component instance
<code>isConnector</code>	Find if instance is connector instance
<code>isPort</code>	Find if instance is port instance

## Examples

### Analyze Latency Characteristics

Create an instantiation for analysis for a system with latency in its wiring. The materials used are copper, fiber, and WiFi.

#### Create Latency Profile with Stereotypes and Properties

Create a System Composer profile with a base, connector, component, and port stereotype. Add properties with default values to each stereotype as needed for analysis.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfileC");
```

Add a base stereotype with properties.

```
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");
```

Add a connector stereotype with properties.

```
connLatency = profile.addStereotype("ConnectorLatency", ...
    Parent="LatencyProfileC.LatencyBase");
connLatency.addProperty("secure", Type="boolean", DefaultValue="true");
connLatency.addProperty("linkDistance", Type="double");
```

Add a component stereotype with properties.

```
nodeLatency = profile.addStereotype("NodeLatency", ...
    Parent="LatencyProfileC.LatencyBase");
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");
```

Add a port stereotype with properties.

```
portLatency = profile.addStereotype("PortLatency", ...
    Parent="LatencyProfileC.LatencyBase");
portLatency.addProperty("queueDepth", Type="double", DefaultValue="4.29");
portLatency.addProperty("dummy", Type="int32");
```

#### Instantiate Using Analysis Function

Create a new model and apply the profile. Create components, ports, and connections in the model. Apply stereotypes to the model elements. Finally, instantiate using the analysis function.

```
model = systemcomposer.createModel("archModel",true);
arch = model.Architecture;
```

Apply profile to model.

```
model.applyProfile("LatencyProfileC");
```

Create components, ports, and connections.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},{'in','out'});

componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower','MotionCommand'});
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},{'in','out'});

c_sensorData = connect(arch,componentSensor,componentPlanning);
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

Clean up the canvas.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

Batch apply stereotypes to model elements.

```
batchApplyStereotype(arch,"Component","LatencyProfileC.NodeLatency");
batchApplyStereotype(arch,"Port","LatencyProfileC.PortLatency");
batchApplyStereotype(arch,"Connector","LatencyProfileC.ConnectorLatency");
```

Instantiate using the analysis function.

```
instance = instantiate(model.Architecture,"LatencyProfileC","NewInstance",...
    Function=@calculateLatency,Arguments="3", ...
    Strict=true,NormalizeUnits=false,Direction="PreOrder")
```

```
instance =
```

```
ArchitectureInstance with properties:
```

```
    Specification: [1x1 systemcomposer.arch.Architecture]
        IsStrict: 1
    NormalizeUnits: 0
    AnalysisFunction: @calculateLatency
    AnalysisDirection: PreOrder
    AnalysisArguments: '3'
    ImmediateUpdate: 0
        Components: [1x3 systemcomposer.analysis.ComponentInstance]
            Ports: [0x0 systemcomposer.analysis.PortInstance]
        Connectors: [1x3 systemcomposer.analysis.ConnectorInstance]
            Name: 'NewInstance'
```

### **Inspect Component, Port, and Connector Instances**

Get properties from component, port, and connector instances.

```
defaultResources = instance.Components(1).getValue("LatencyProfileC.NodeLatency.resources")
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue("LatencyProfileC.ConnectorLatency.secure")
defaultSecure = logical
1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue("LatencyProfileC.PortLatency.queueDepth")
defaultQueueDepth = 4.2900
```

## Battery Sizing and Automotive Electrical System Analysis

### Overview

Model a typical automotive electrical system as an architectural model and run a primitive analysis. The elements in the model can be broadly grouped as either a source or a load. Various properties of the sources and loads are set as part of the stereotype. This example uses the `iterate` method of the specification API to iterate through each element of the model and run analysis using the stereotype properties.

### Structure of Model

The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set on each component, the following analyses are performed:

- Total KeyOffLoad.
- Number of days required for KeyOffLoad to discharge 30% of the battery.
- Total CrankingInRush current.
- Total Cranking current.
- Ability of the battery to start the vehicle at 0°F based on the battery cold cranking amps (CCA). The discharge time is computed based on Puekert coefficient (k), which describes the relationship between the rate of discharge and the available capacity of the battery.

### Load Model and Run Analysis

```
scExampleAutomotiveElectricalSystemAnalysis
archModel = systemcomposer.loadModel('scExampleAutomotiveElectricalSystemAnalysis');
```

Instantiate battery sizing class used by the analysis function to store analysis results.

```
objcomputeBatterySizing = computeBatterySizing;
```

Run the analysis using the iterator.

```
archModel.iterate('Topdown',@computeLoad,objcomputeBatterySizing)
```

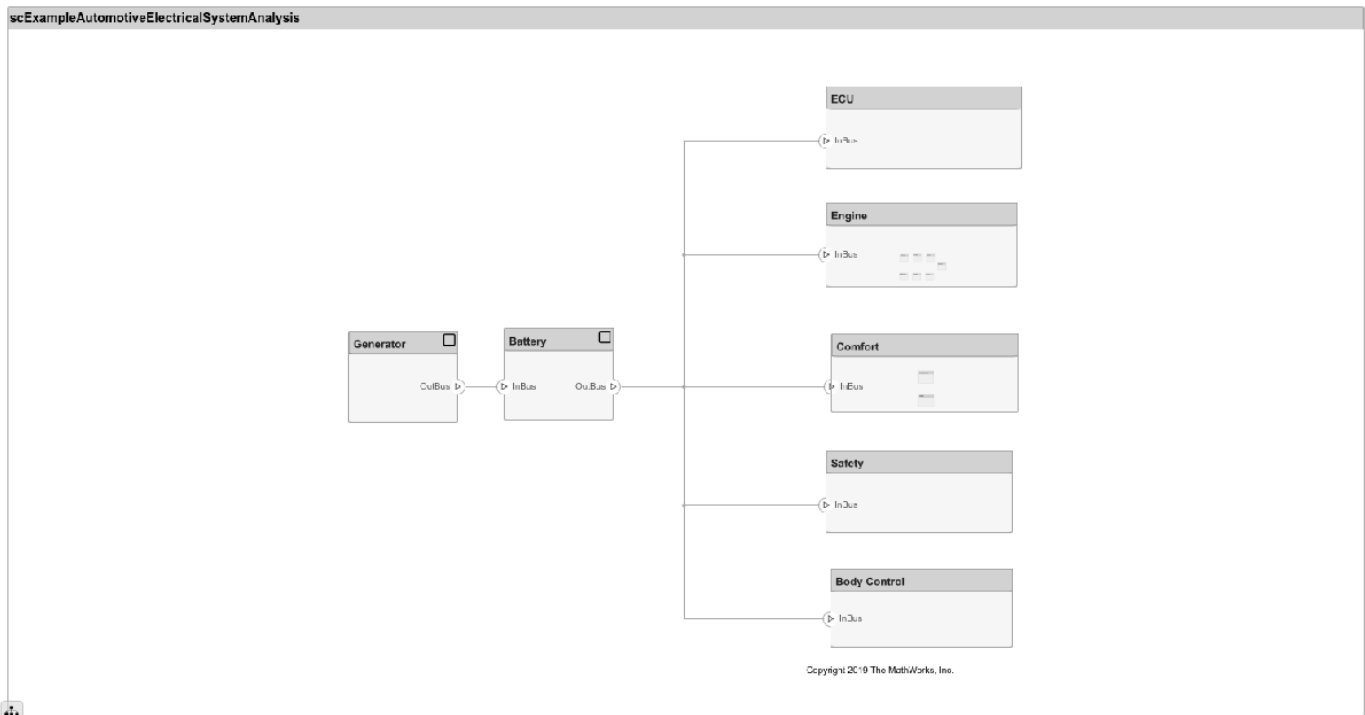
Display analysis results.

```
objcomputeBatterySizing.displayResults
```

```
Total KeyOffLoad: 158.708 mA
Number of days required for KeyOffLoad to discharge 30% of battery: 55.789.
```

Total CrankingInRush current: 70 A  
Total Cranking current: 104 A  
CCA of the specified battery is sufficient to start the car at 0 F.

```
ans =  
  computeBatterySizing with properties:  
    totalCrankingInrushCurrent: 70  
    totalCrankingCurrent: 104  
    totalAccesoriesCurrent: 71.6667  
    totalKeyOffLoad: 158.7080  
    batteryCCA: 500  
    batteryCapacity: 850  
    puekertcoefficient: 1.2000
```



**Close Model**

```
bdclose('scExampleAutomotiveElectricalSystemAnalysis');
```

**More About****Definitions**

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

deleteInstance | update | refresh | save | instantiate | loadInstance | iterate |  
systemcomposer.analysis.ArchitectureInstance |  
systemcomposer.analysis.PortInstance |  
systemcomposer.analysis.ConnectorInstance | systemcomposer.analysis.Instance

### Topics

"Write Analysis Function"

# systemcomposer.analysis.ConnectorInstance

Connector in analysis instance

## Description

A ConnectorInstance object represents an instance of a connector.

## Creation

Create an instance of an architecture using the `instantiate` function.

```
instance = instantiate(model.Architecture, 'LatencyProfile', 'NewInstance', ...  
'Function', @calculateLatency, 'Arguments', '3', 'Strict', true, ...  
'NormalizeUnits', false, 'Direction', 'PreOrder')
```

## Properties

### Name — Name of instance

character vector

Name of instance, specified as a character vector.

Example: 'NewInterface'

Data Types: char

### Parent — Component that contains connector

component instance object

Component that contains connector, specified as a `systemcomposer.analysis.ComponentInstance` object.

### Ports — Ports of connector instance

array of port instance objects

Ports of connector instance, specified as an array of `systemcomposer.analysis.PortInstance` objects.

### SourcePort — Source port instance

port instance object

Source port instance, specified as a `systemcomposer.analysis.PortInstance` object.

### DestinationPort — Destination port instance

port instance object

Destination port instance, specified as a `systemcomposer.analysis.PortInstance` object.

### Specification — Reference to connector in design model

connector object | physical connector object



Reference to connector in design model, specified as a `systemcomposer.arch.Connector` or `systemcomposer.arch.PhysicalConnector` object.

### QualifiedName — Qualified name of connector

character vector

Qualified name of connector, specified as a character vector of the form '`<PathToSourceComponent>:<PortDirection>-<PathToDestinationComponent>:<PortDirection>`'.

Example: `'model2:In->model2/Component:In'`

Data Types: `char`

## Object Functions

<code>getValue</code>	Get value of property from element instance
<code>setValue</code>	Set value of property for element instance
<code>hasValue</code>	Find if element instance has property value
<code>isArchitecture</code>	Find if instance is architecture instance
<code>isComponent</code>	Find if instance is component instance
<code>isConnector</code>	Find if instance is connector instance
<code>isPort</code>	Find if instance is port instance

## Examples

### Analyze Latency Characteristics

Create an instantiation for analysis for a system with latency in its wiring. The materials used are copper, fiber, and WiFi.

### Create Latency Profile with Stereotypes and Properties

Create a System Composer profile with a base, connector, component, and port stereotype. Add properties with default values to each stereotype as needed for analysis.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfileC");
```

Add a base stereotype with properties.

```
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");
```

Add a connector stereotype with properties.

```
connLatency = profile.addStereotype("ConnectorLatency",...
    Parent="LatencyProfileC.LatencyBase");
connLatency.addProperty("secure",Type="boolean",DefaultValue="true");
connLatency.addProperty("linkDistance",Type="double");
```

Add a component stereotype with properties.

```
nodeLatency = profile.addStereotype("NodeLatency",...
    Parent="LatencyProfileC.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");
```

Add a port stereotype with properties.

```
portLatency = profile.addStereotype("PortLatency", ...
    Parent="LatencyProfileC.LatencyBase");
portLatency.addProperty("queueDepth", Type="double", DefaultValue="4.29");
portLatency.addProperty("dummy", Type="int32");
```

### Instantiate Using Analysis Function

Create a new model and apply the profile. Create components, ports, and connections in the model. Apply stereotypes to the model elements. Finally, instantiate using the analysis function.

```
model = systemcomposer.createModel("archModel", true);
arch = model.Architecture;
```

Apply profile to model.

```
model.applyProfile("LatencyProfileC");
```

Create components, ports, and connections.

```
componentSensor = addComponent(arch, "Sensor");
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, {'in', 'out'});

componentPlanning = addComponent(arch, "Planning");
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorPower', 'MotionCommand'});
componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, {'in', 'out'});
```

```
c_sensorData = connect(arch, componentSensor, componentPlanning);
c_motionData = connect(arch, componentMotion, componentSensor);
c_motionCommand = connect(arch, componentPlanning, componentMotion);
```

Clean up the canvas.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

Batch apply stereotypes to model elements.

```
batchApplyStereotype(arch, "Component", "LatencyProfileC.NodeLatency");
batchApplyStereotype(arch, "Port", "LatencyProfileC.PortLatency");
batchApplyStereotype(arch, "Connector", "LatencyProfileC.ConnectorLatency");
```

Instantiate using the analysis function.

```
instance = instantiate(model.Architecture, "LatencyProfileC", "NewInstance", ...
    Function=@calculateLatency, Arguments="3", ...
    Strict=true, NormalizeUnits=false, Direction="PreOrder")
```

```
instance =
    ArchitectureInstance with properties:
```

```
    Specification: [1x1 systemcomposer.arch.Architecture]
    IsStrict: 1
    NormalizeUnits: 0
    AnalysisFunction: @calculateLatency
    AnalysisDirection: PreOrder
    AnalysisArguments: '3'
    ImmediateUpdate: 0
```

```

Components: [1x3 systemcomposer.analysis.ComponentInstance]
Ports: [0x0 systemcomposer.analysis.PortInstance]
Connectors: [1x3 systemcomposer.analysis.ConnectorInstance]
Name: 'NewInstance'

```

## Inspect Component, Port, and Connector Instances

Get properties from component, port, and connector instances.

```
defaultResources = instance.Components(1).getValue("LatencyProfileC.NodeLatency.resources")
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue("LatencyProfileC.ConnectorLatency.secure")
```

```
defaultSecure = logical
1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue("LatencyProfileC.PortLatency.queueDepth")
```

```
defaultQueueDepth = 4.2900
```

## Battery Sizing and Automotive Electrical System Analysis

### Overview

Model a typical automotive electrical system as an architectural model and run a primitive analysis. The elements in the model can be broadly grouped as either a source or a load. Various properties of the sources and loads are set as part of the stereotype. This example uses the `iterate` method of the specification API to iterate through each element of the model and run analysis using the stereotype properties.

### Structure of Model

The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set on each component, the following analyses are performed:

- Total KeyOffLoad.
- Number of days required for KeyOffLoad to discharge 30% of the battery.
- Total CrankingInRush current.
- Total Cranking current.
- Ability of the battery to start the vehicle at 0°F based on the battery cold cranking amps (CCA). The discharge time is computed based on Puekert coefficient (k), which describes the relationship between the rate of discharge and the available capacity of the battery.

### Load Model and Run Analysis

```

scExampleAutomotiveElectricalSystemAnalysis
archModel = systemcomposer.loadModel('scExampleAutomotiveElectricalSystemAnalysis');

```

Instantiate battery sizing class used by the analysis function to store analysis results.

```
objcomputeBatterySizing = computeBatterySizing;
```

Run the analysis using the iterator.

```
archModel.iterate('Topdown',@computeLoad,objcomputeBatterySizing)
```

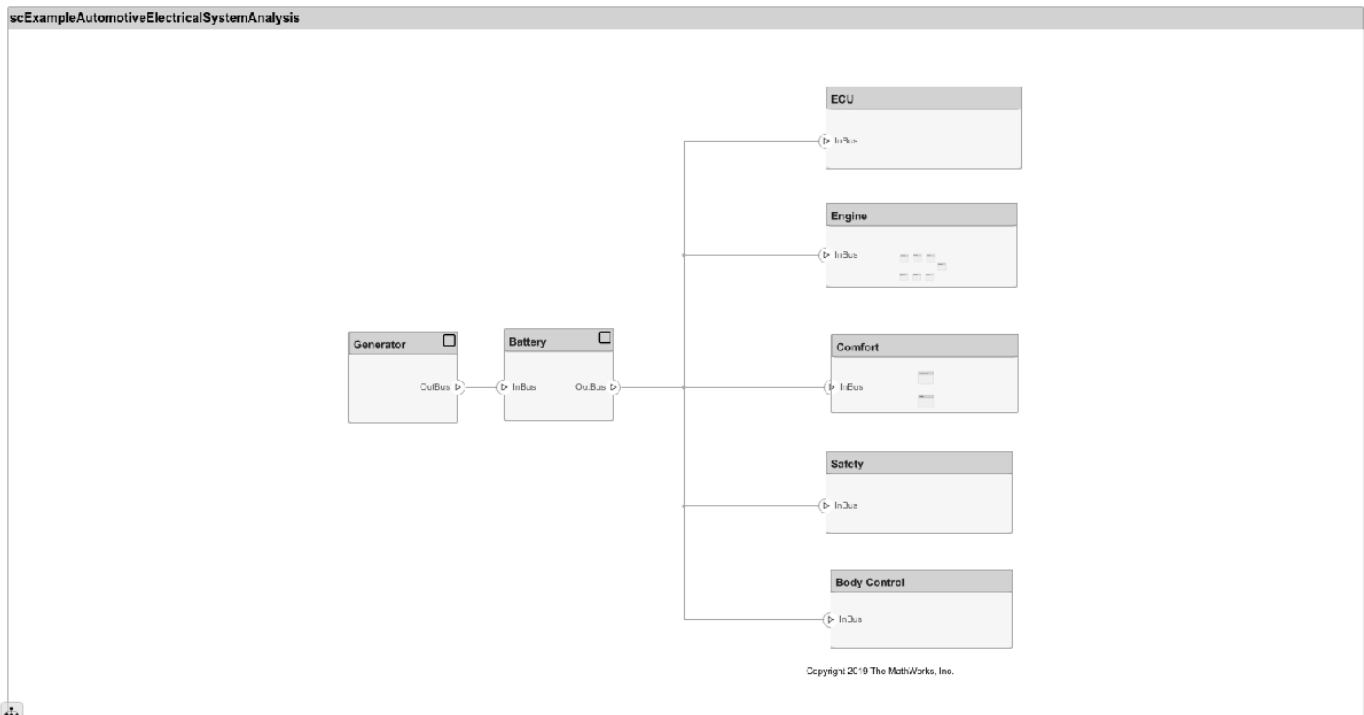
Display analysis results.

```
objcomputeBatterySizing.displayResults
```

```
Total KeyOffLoad: 158.708 mA
Number of days required for KeyOffLoad to discharge 30% of battery: 55.789.
Total CrankingInRush current: 70 A
Total Cranking current: 104 A
CCA of the specified battery is sufficient to start the car at 0 F.
```

```
ans =
  computeBatterySizing with properties:

    totalCrankingInrushCurrent: 70
      totalCrankingCurrent: 104
        totalAccesoriesCurrent: 71.6667
          totalKeyOffLoad: 158.7080
            batteryCCA: 500
              batteryCapacity: 850
                puekertcoefficient: 1.2000
```



**Close Model**

```
bdclose('scExampleAutomotiveElectricalSystemAnalysis');
```

**More About****Definitions**

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

deleteInstance | update | refresh | save | instantiate | loadInstance | iterate |  
systemcomposer.analysis.PortInstance |  
systemcomposer.analysis.ArchitectureInstance |  
systemcomposer.analysis.ComponentInstance | systemcomposer.analysis.Instance

### Topics

"Write Analysis Function"

## systemcomposer.analysis.Instance

Element in analysis instance

### Description

An Instance object represents an instance of a System Composer model element.

Related objects include:

- `systemcomposer.analysis.ArchitectureInstance`
- `systemcomposer.analysis.ComponentInstance`
- `systemcomposer.analysis.PortInstance`
- `systemcomposer.analysis.ConnectorInstance`

### Creation

Create an instance of an architecture using the `instantiate` function.

```
instance = instantiate(model.Architecture, 'LatencyProfile', 'NewInstance', ...  
'Function', @calculateLatency, 'Arguments', '3', 'Strict', true, ...  
'NormalizeUnits', false, 'Direction', 'PreOrder')
```

### Properties

#### Name — Name of instance

character vector

Name of instance, specified as a character vector.

Example: 'NewInstance'

Data Types: char

### Object Functions

<code>getValue</code>	Get value of property from element instance
<code>setValue</code>	Set value of property for element instance
<code>hasValue</code>	Find if element instance has property value
<code>isArchitecture</code>	Find if instance is architecture instance
<code>isComponent</code>	Find if instance is component instance
<code>isConnector</code>	Find if instance is connector instance
<code>isPort</code>	Find if instance is port instance

### Examples



## Analyze Latency Characteristics

Create an instantiation for analysis for a system with latency in its wiring. The materials used are copper, fiber, and WiFi.

### Create Latency Profile with Stereotypes and Properties

Create a System Composer profile with a base, connector, component, and port stereotype. Add properties with default values to each stereotype as needed for analysis.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfileC");
```

Add a base stereotype with properties.

```
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");
```

Add a connector stereotype with properties.

```
connLatency = profile.addStereotype("ConnectorLatency",...
    Parent="LatencyProfileC.LatencyBase");
connLatency.addProperty("secure",Type="boolean",DefaultValue="true");
connLatency.addProperty("linkDistance",Type="double");
```

Add a component stereotype with properties.

```
nodeLatency = profile.addStereotype("NodeLatency",...
    Parent="LatencyProfileC.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");
```

Add a port stereotype with properties.

```
portLatency = profile.addStereotype("PortLatency",...
    Parent="LatencyProfileC.LatencyBase");
portLatency.addProperty("queueDepth",Type="double",DefaultValue="4.29");
portLatency.addProperty("dummy",Type="int32");
```

### Instantiate Using Analysis Function

Create a new model and apply the profile. Create components, ports, and connections in the model. Apply stereotypes to the model elements. Finally, instantiate using the analysis function.

```
model = systemcomposer.createModel("archModel",true);
arch = model.Architecture;
```

Apply profile to model.

```
model.applyProfile("LatencyProfileC");
```

Create components, ports, and connections.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},{'in','out'});

componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower','MotionCommand'});
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

```
c_sensorData = connect(arch,componentSensor,componentPlanning);
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

Clean up the canvas.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

Batch apply stereotypes to model elements.

```
batchApplyStereotype(arch,"Component","LatencyProfileC.NodeLatency");
batchApplyStereotype(arch,"Port","LatencyProfileC.PortLatency");
batchApplyStereotype(arch,"Connector","LatencyProfileC.ConnectorLatency");
```

Instantiate using the analysis function.

```
instance = instantiate(model.Architecture,"LatencyProfileC","NewInstance",...
    Function=@calculateLatency,Arguments="3", ...
    Strict=true,NormalizeUnits=false,Direction="PreOrder")
```

```
instance =
  ArchitectureInstance with properties:

    Specification: [1x1 systemcomposer.arch.Architecture]
      IsStrict: 1
    NormalizeUnits: 0
    AnalysisFunction: @calculateLatency
    AnalysisDirection: PreOrder
    AnalysisArguments: '3'
    ImmediateUpdate: 0
      Components: [1x3 systemcomposer.analysis.ComponentInstance]
        Ports: [0x0 systemcomposer.analysis.PortInstance]
      Connectors: [1x3 systemcomposer.analysis.ConnectorInstance]
        Name: 'NewInstance'
```

### Inspect Component, Port, and Connector Instances

Get properties from component, port, and connector instances.

```
defaultResources = instance.Components(1).getValue("LatencyProfileC.NodeLatency.resources")
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue("LatencyProfileC.ConnectorLatency.secure")
```

```
defaultSecure = logical
  1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue("LatencyProfileC.PortLatency.queueDepth")
```

```
defaultQueueDepth = 4.2900
```

## Battery Sizing and Automotive Electrical System Analysis

### Overview

Model a typical automotive electrical system as an architectural model and run a primitive analysis. The elements in the model can be broadly grouped as either a source or a load. Various properties of the sources and loads are set as part of the stereotype. This example uses the `iterate` method of the specification API to iterate through each element of the model and run analysis using the stereotype properties.

### Structure of Model

The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set on each component, the following analyses are performed:

- Total KeyOffLoad.
- Number of days required for KeyOffLoad to discharge 30% of the battery.
- Total CrankingInRush current.
- Total Cranking current.
- Ability of the battery to start the vehicle at 0°F based on the battery cold cranking amps (CCA). The discharge time is computed based on Puekert coefficient (k), which describes the relationship between the rate of discharge and the available capacity of the battery.

### Load Model and Run Analysis

```
scExampleAutomotiveElectricalSystemAnalysis
archModel = systemcomposer.loadModel('scExampleAutomotiveElectricalSystemAnalysis');
```

Instantiate battery sizing class used by the analysis function to store analysis results.

```
objcomputeBatterySizing = computeBatterySizing;
```

Run the analysis using the iterator.

```
archModel.iterate('Topdown',@computeLoad,objcomputeBatterySizing)
```

Display analysis results.

```
objcomputeBatterySizing.displayResults
```

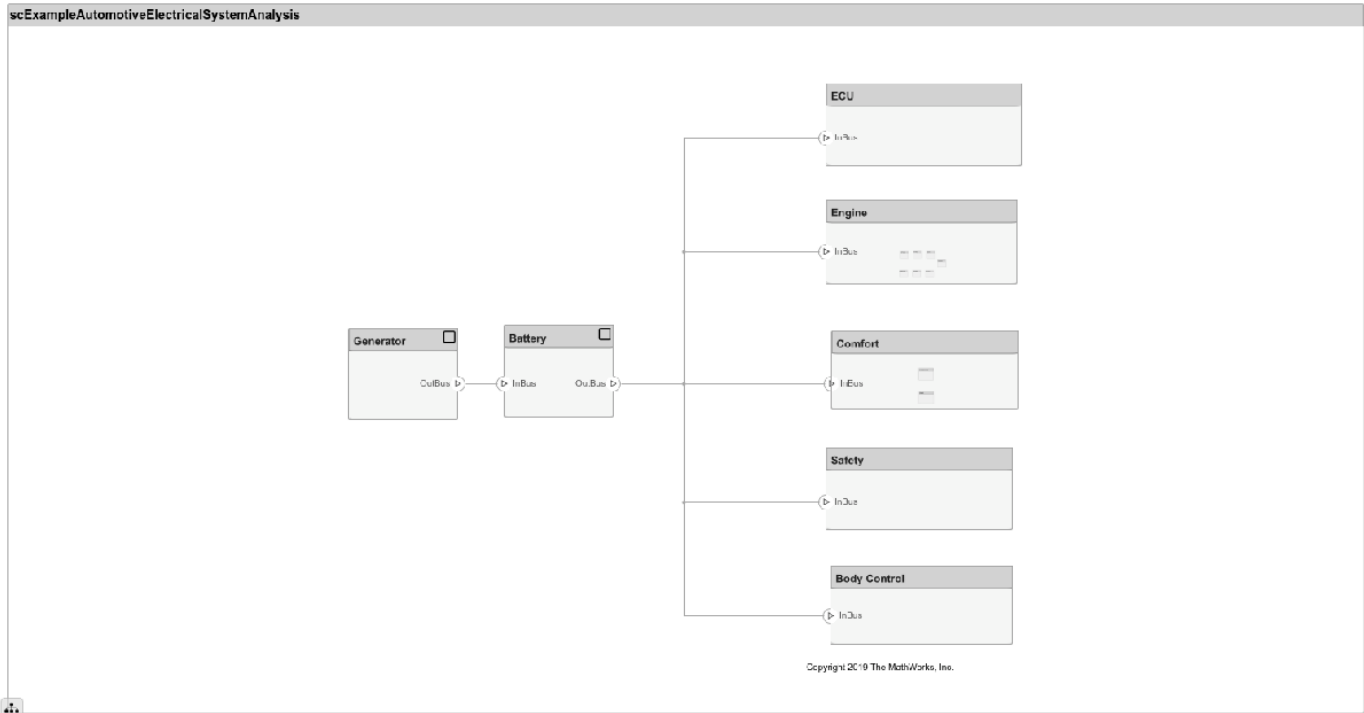
```
Total KeyOffLoad: 158.708 mA
Number of days required for KeyOffLoad to discharge 30% of battery: 55.789.
Total CrankingInRush current: 70 A
Total Cranking current: 104 A
CCA of the specified battery is sufficient to start the car at 0 F.
```

```
ans =
```

```
computeBatterySizing with properties:
```

```
totalCrankingInrushCurrent: 70
totalCrankingCurrent: 104
totalAccesoriesCurrent: 71.6667
totalKeyOffLoad: 158.7080
batteryCCA: 500
```

```
batteryCapacity: 850
puekertcoefficient: 1.2000
```



### Close Model

```
bdclose('scExampleAutomotiveElectricalSystemAnalysis');
```

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>

Term	Definition	Application	More Information
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>• “Analysis Function Constructs”</li> <li>• “Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	<p>Perform operations on a model:</p> <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> <p>A System Composer model is stored as an SLX file.</p>	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

deleteInstance | instantiate | loadInstance | save | update | refresh | iterate |  
systemcomposer.analysis.ArchitectureInstance |  
systemcomposer.analysis.ComponentInstance |  
systemcomposer.analysis.PortInstance |  
systemcomposer.analysis.ConnectorInstance

### Topics

"Write Analysis Function"

# systemcomposer.analysis.PortInstance

Port in analysis instance

## Description

A PortInstance object represents an instance of a port.

## Creation

Create an instance of an architecture using the `instantiate` function.

```
instance = instantiate(model.Architecture, 'LatencyProfile', 'NewInstance', ...
    'Function', @calculateLatency, 'Arguments', '3', 'Strict', true, ...
    'NormalizeUnits', false, 'Direction', 'PreOrder')
```

## Properties

### Name — Name of instance

character vector

Name of instance, specified as a character vector.

Example: 'NewInstance'

Data Types: char

### Parent — Component that contains port

component instance object

Component that contains port, specified as a `systemcomposer.analysis.ComponentInstance` object.

### Specification — Reference to port in design model

base port object

Reference to port in design model, specified as a `systemcomposer.arch.BasePort` object.

### QualifiedName — Qualified name of port

character vector

Qualified name of port, specified as a character vector of the form '<PathToComponent>:<PortDirection>'.

Example: 'model/Component:In'

Data Types: char

### Incoming — Incoming connection

connector instance object

Incoming connection, specified as a `systemcomposer.analysis.ConnectorInstance` object.

**Outgoing — Outgoing connection**

connector instance object

Outgoing connection, specified as a `systemcomposer.analysis.ConnectorInstance` object.

**Object Functions**

<code>getValue</code>	Get value of property from element instance
<code>setValue</code>	Set value of property for element instance
<code>hasValue</code>	Find if element instance has property value
<code>isArchitecture</code>	Find if instance is architecture instance
<code>isComponent</code>	Find if instance is component instance
<code>isConnector</code>	Find if instance is connector instance
<code>isPort</code>	Find if instance is port instance

**Examples****Analyze Latency Characteristics**

Create an instantiation for analysis for a system with latency in its wiring. The materials used are copper, fiber, and WiFi.

**Create Latency Profile with Stereotypes and Properties**

Create a System Composer profile with a base, connector, component, and port stereotype. Add properties with default values to each stereotype as needed for analysis.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfileC");
```

Add a base stereotype with properties.

```
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");
```

Add a connector stereotype with properties.

```
connLatency = profile.addStereotype("ConnectorLatency",...
    Parent="LatencyProfileC.LatencyBase");
connLatency.addProperty("secure",Type="boolean",DefaultValue="true");
connLatency.addProperty("linkDistance",Type="double");
```

Add a component stereotype with properties.

```
nodeLatency = profile.addStereotype("NodeLatency",...
    Parent="LatencyProfileC.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");
```

Add a port stereotype with properties.

```
portLatency = profile.addStereotype("PortLatency",...
    Parent="LatencyProfileC.LatencyBase");
portLatency.addProperty("queueDepth",Type="double",DefaultValue="4.29");
portLatency.addProperty("dummy",Type="int32");
```



## Instantiate Using Analysis Function

Create a new model and apply the profile. Create components, ports, and connections in the model. Apply stereotypes to the model elements. Finally, instantiate using the analysis function.

```
model = systemcomposer.createModel("archModel", true);
arch = model.Architecture;
```

Apply profile to model.

```
model.applyProfile("LatencyProfileC");
```

Create components, ports, and connections.

```
componentSensor = addComponent(arch, "Sensor");
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, {'in', 'out'});

componentPlanning = addComponent(arch, "Planning");
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorPower', 'MotionCommand'});
componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, {'in', 'out'});

c_sensorData = connect(arch, componentSensor, componentPlanning);
c_motionData = connect(arch, componentMotion, componentSensor);
c_motionCommand = connect(arch, componentPlanning, componentMotion);
```

Clean up the canvas.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

Batch apply stereotypes to model elements.

```
batchApplyStereotype(arch, "Component", "LatencyProfileC.NodeLatency");
batchApplyStereotype(arch, "Port", "LatencyProfileC.PortLatency");
batchApplyStereotype(arch, "Connector", "LatencyProfileC.ConnectorLatency");
```

Instantiate using the analysis function.

```
instance = instantiate(model.Architecture, "LatencyProfileC", "NewInstance", ...
    Function=@calculateLatency, Arguments="3", ...
    Strict=true, NormalizeUnits=false, Direction="PreOrder")
```

```
instance =
```

```
ArchitectureInstance with properties:
```

```
    Specification: [1x1 systemcomposer.arch.Architecture]
        IsStrict: 1
    NormalizeUnits: 0
    AnalysisFunction: @calculateLatency
    AnalysisDirection: PreOrder
    AnalysisArguments: '3'
    ImmediateUpdate: 0
        Components: [1x3 systemcomposer.analysis.ComponentInstance]
            Ports: [0x0 systemcomposer.analysis.PortInstance]
        Connectors: [1x3 systemcomposer.analysis.ConnectorInstance]
            Name: 'NewInstance'
```

### Inspect Component, Port, and Connector Instances

Get properties from component, port, and connector instances.

```
defaultResources = instance.Components(1).getValue("LatencyProfileC.NodeLatency.resources")
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue("LatencyProfileC.ConnectorLatency.secure")
```

```
defaultSecure = logical  
1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue("LatencyProfileC.PortLatency.queueDepth")
```

```
defaultQueueDepth = 4.2900
```

### Battery Sizing and Automotive Electrical System Analysis

#### Overview

Model a typical automotive electrical system as an architectural model and run a primitive analysis. The elements in the model can be broadly grouped as either a source or a load. Various properties of the sources and loads are set as part of the stereotype. This example uses the `iterate` method of the specification API to iterate through each element of the model and run analysis using the stereotype properties.

#### Structure of Model

The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set on each component, the following analyses are performed:

- Total KeyOffLoad.
- Number of days required for KeyOffLoad to discharge 30% of the battery.
- Total CrankingInRush current.
- Total Cranking current.
- Ability of the battery to start the vehicle at 0°F based on the battery cold cranking amps (CCA). The discharge time is computed based on Puekert coefficient (k), which describes the relationship between the rate of discharge and the available capacity of the battery.

#### Load Model and Run Analysis

```
scExampleAutomotiveElectricalSystemAnalysis  
archModel = systemcomposer.loadModel('scExampleAutomotiveElectricalSystemAnalysis');
```

Instantiate battery sizing class used by the analysis function to store analysis results.

```
objcomputeBatterySizing = computeBatterySizing;
```

Run the analysis using the iterator.

```
archModel.iterate('Topdown',@computeLoad,objcomputeBatterySizing)
```

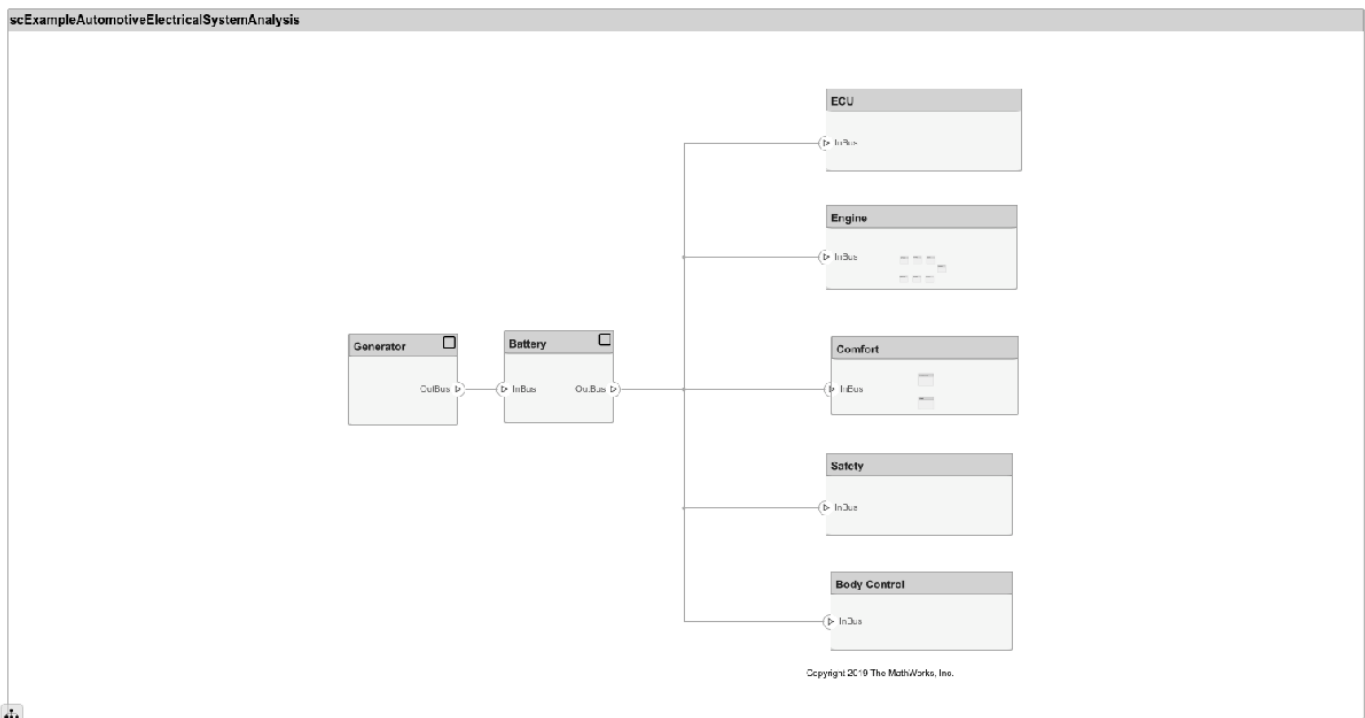
Display analysis results.

```
objcomputeBatterySizing.displayResults
```

```
Total KeyOffLoad: 158.708 mA
Number of days required for KeyOffLoad to discharge 30% of battery: 55.789.
Total CrankingInRush current: 70 A
Total Cranking current: 104 A
CCA of the specified battery is sufficient to start the car at 0 F.
```

```
ans =
  computeBatterySizing with properties:

    totalCrankingInrushCurrent: 70
      totalCrankingCurrent: 104
    totalAccesoriesCurrent: 71.6667
      totalKeyOffLoad: 158.7080
    batteryCCA: 500
    batteryCapacity: 850
    puekertcoefficient: 1.2000
```



**Close Model**

```
bdclose('scExampleAutomotiveElectricalSystemAnalysis');
```

**More About**

**Definitions**

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>• “Analyze Architecture Model with Analysis Function”</li> <li>• “Analyze Architecture”</li> <li>• “Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>• “Analysis Function Constructs”</li> <li>• “Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	<p>Perform operations on a model:</p> <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> <p>A System Composer model is stored as an SLX file.</p>	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

deleteInstance | update | refresh | save | instantiate | loadInstance | iterate |  
systemcomposer.analysis.ConnectorInstance |  
systemcomposer.analysis.ComponentInstance |  
systemcomposer.analysis.ArchitectureInstance | systemcomposer.analysis.Instance

### Topics

"Write Analysis Function"

# systemcomposer.arch.Architecture

Architecture in model

## Description

The `Architecture` object represents the architecture in a System Composer model. This class is derived from `systemcomposer.arch.Element`.

## Creation

Create a model using the `systemcomposer.createModel` function and get the root architecture using the `Architecture` property on the `systemcomposer.arch.Model` object.

```
model = systemcomposer.createModel('archModel');  
arch = get(model, 'Architecture');
```

## Properties

### Name — Name of architecture

character vector

Name of architecture, specified as a character vector. The architecture name is derived from the parent component or model name to which the architecture belongs.

Example: 'archModel'

Data Types: char

### Definition — Definition type of architecture

`ArchitectureDefinition` enumeration

Definition type of architecture, specified as `composition`, `behavior`, or `view`.

Data Types: enum

### Parent — Parent component

component object

Parent component that owns architecture, specified as a `systemcomposer.arch.Component` object.

### Components — Child components

array of component objects

Child components of architecture, specified as an array of `systemcomposer.arch.Component` objects.

### Ports — Architecture ports

array of architecture port objects

Architecture ports, specified as an array of `systemcomposer.arch.ArchitecturePort` objects.

**Connectors — Connectors that connect child components of architecture**

array of connector objects

Connectors that connect child components of architecture, specified as an array of `systemcomposer.arch.Connector` or `systemcomposer.arch.PhysicalConnector` objects.

**Parameters — Parameters of component**

array of parameter objects

Parameters of component, specified as an array of `systemcomposer.arch.Parameter` objects.

**UUID — Universal unique identifier**

character vector

Universal unique identifier for architecture, specified as a character vector.

Example: `'91d5de2c-b14c-4c76-a5d6-5dd0037c52df'`

Data Types: `char`

**ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the architecture and through all operations that preserve the UUID.

Data Types: `char`

**Model — Parent model**

model object

Parent System Composer model of architecture, specified as a `systemcomposer.arch.Model` object.

**SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox™ programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: `double`

**SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: `double`



## Object Functions

addComponent	Add components to architecture
addVariantComponent	Add variant components to architecture
addPort	Add ports to architecture
addFunction	Add functions to architecture of software component
addParameter	Add parameter to architecture
getParameter	Get parameter from architecture or component
connect	Create architecture model connections
applyStereotype	Apply stereotype to architecture model element
getStereotypes	Get stereotypes applied on element of architecture model
removeStereotype	Remove stereotype from model element
batchApplyStereotype	Apply stereotype to all elements in architecture
iterate	Iterate over model elements
instantiate	Create analysis instance from specification
setProperty	Set property value corresponding to stereotype applied to element
getProperty	Get property value corresponding to stereotype applied to element
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from element
getStereotypeProperties	Get stereotype property names on element
removeProfile	Remove profile from model
applyProfile	Apply profile to model
hasStereotype	Find if element has stereotype applied
hasProperty	Find if element has property
getEvaluatedParameterValue	Get evaluated value of parameter from element
getParameterNames	Get parameter names on element
getParameterValue	Get value of parameter
setParameterValue	Set value of parameter
setUnit	Set units on parameter value
resetParameterToDefault	Reset parameter on component to default value

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the

value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
SensorInterfaces.sldd				
GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

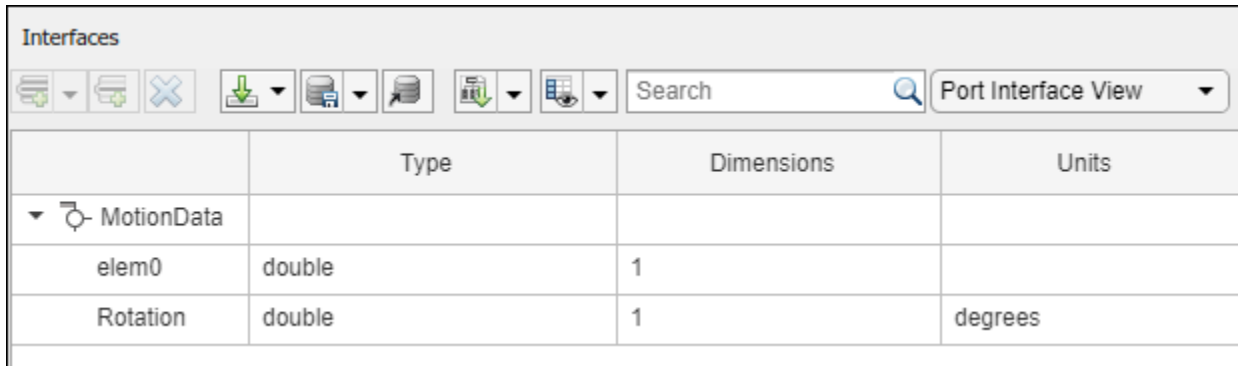
```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)
```



```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces			
 <input type="text" value="Search"/> <span>Port Interface View</span>			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch,"Command","in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

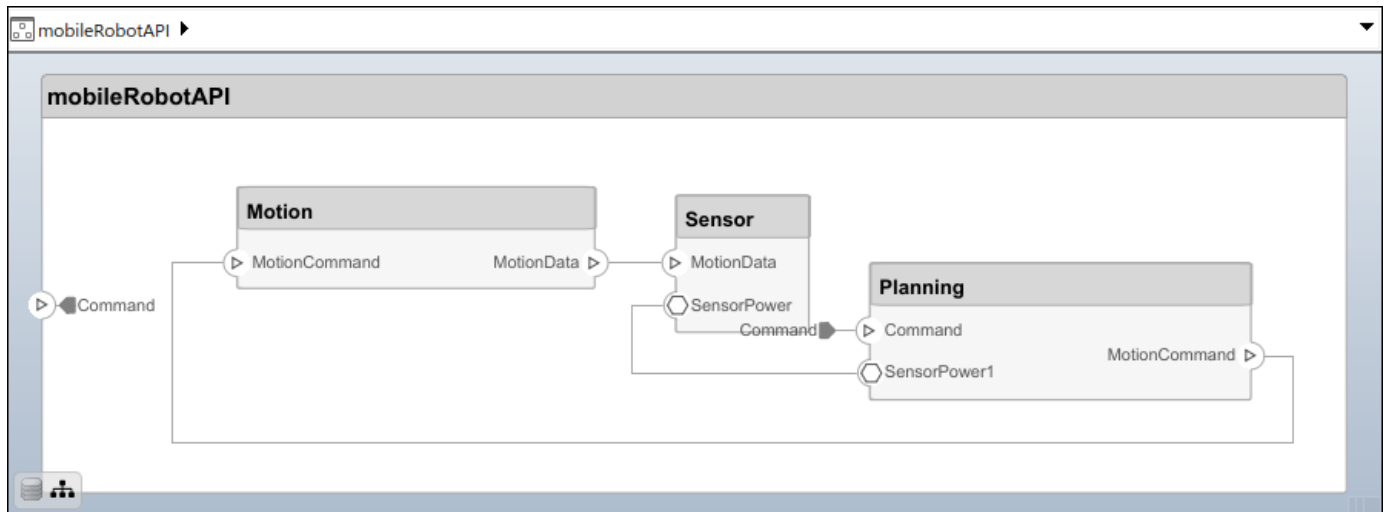
```
compPort = getPort(componentPlanning,"Command");
c_Command = connect(archPort,compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

#### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

#### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
```

```

addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");

```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```

applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")

```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```

batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");

```

Set properties for each component.

```

setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
  'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
  'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
  'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

## Add Hierarchy

Add two components named `Controller` and `Scope` inside the `Motion` component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

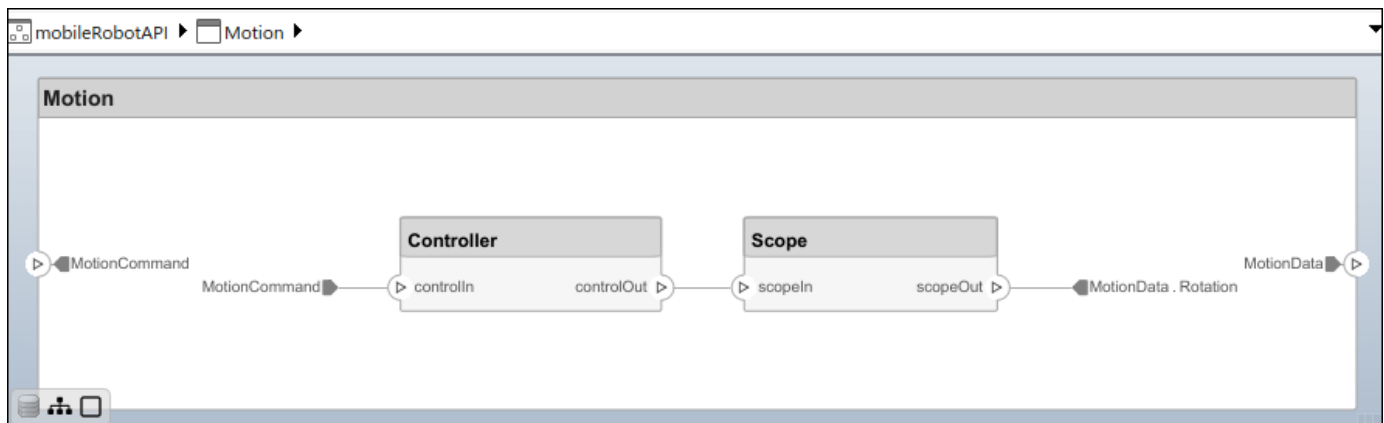
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

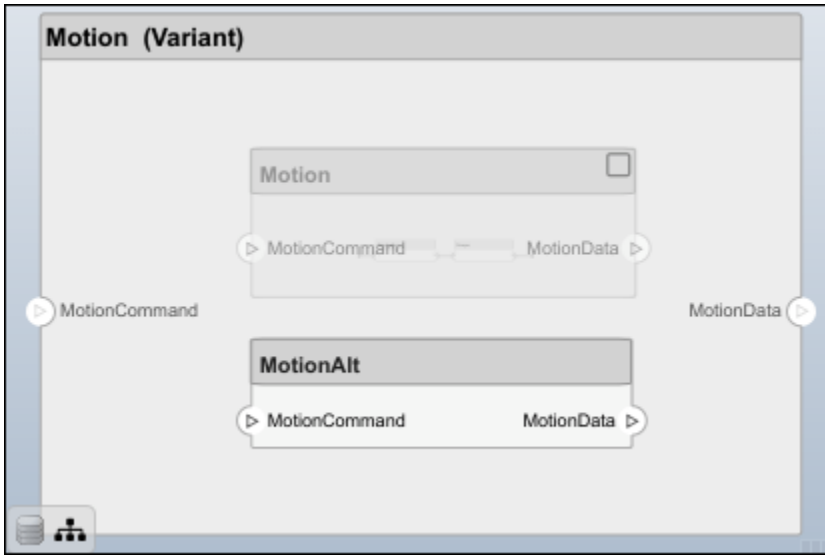
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>



Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## **Version History**

**Introduced in R2019a**

### **See Also**

`systemcomposer.arch.Component` | `systemcomposer.arch.Element` | `Component`

### **Topics**

“Create Architecture Model”

# systemcomposer.arch.ArchitecturePort

Architecture port

## Description

An ArchitecturePort object represents the input, output, and physical ports of a System Composer architecture. This class inherits from `systemcomposer.arch.BasePort`. This class is derived from `systemcomposer.arch.Element`.

## Creation

Create an architecture port using the `addPort` function.

```
port = addPort(architecture, 'in')
```

## Properties

### Name — Name of port

character vector

Name of port, specified as a character vector.

Example: 'newPort'

Data Types: char

### Direction — Port direction

'Input' | 'Output' | 'Physical' | 'Client' | 'Server'

Port direction, specified as a character vector.

Data Types: char

### InterfaceName — Name of interface associated with port

character vector

Name of interface associated with port, specified as a character vector.

Data Types: char

### Interface — Interface associated with port

data interface object | value type object | physical interface object | service interface object

Interface associated with port, specified as a `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### Connectors — Port connectors

array of connector objects

Port connectors, specified as an array of `systemcomposer.arch.Connector` or `systemcomposer.arch.PhysicalConnector` objects.

**Connected — Whether port has connections**

true or 1 | false or 0

Whether port has connections, specified as a logical.

Data Types: logical

**Parent — Architecture that owns port**

architecture object

Architecture that owns port, specified as a `systemcomposer.arch.Architecture` object.

**UUID — Universal unique identifier**

character vector

Universal unique identifier for architecture port, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the architecture port and through all operations that preserve the UUID.

Data Types: char

**Model — Parent model**

model object

Parent System Composer model of architecture port, specified as a `systemcomposer.arch.Model` object.

**SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: double

**SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: double

## Object Functions

connect	Create architecture model connections
setName	Set name for port
setInterface	Set interface for port
createInterface	Create and set owned interface for port
makeOwnedInterfaceShared	Convert owned interface to shared interface
applyStereotype	Apply stereotype to architecture model element
getStereotypes	Get stereotypes applied on element of architecture model
removeStereotype	Remove stereotype from model element
setProperty	Set property value corresponding to stereotype applied to element
getProperty	Get property value corresponding to stereotype applied to element
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from element
getStereotypeProperties	Get stereotype property names on element
hasStereotype	Find if element has stereotype applied
hasProperty	Find if element has property
getQualifiedName	Get model element qualified name
destroy	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
```

```

physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface, "ElectricalElement", ...
    Type="electrical.electrical");
linkDictionary(model, "SensorInterfaces.sldd");

```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sldd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```

componentSensor = addComponent(arch, "Sensor");
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, ...
    {'in', 'physical'});
sensorPorts(2).setInterface(physicalInterface)

```

```

componentPlanning = addComponent(arch, "Planning");
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorPower1', 'MotionCommand'}, ...
    {'in', 'physical', 'out'});
planningPorts(2).setInterface(physicalInterface)

```

```

componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, ...
    {'in', 'out'});

```




Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```

ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");

```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.

Interfaces			
			
<input type="text" value="Search"/>  <span>Port Interface View</span>			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

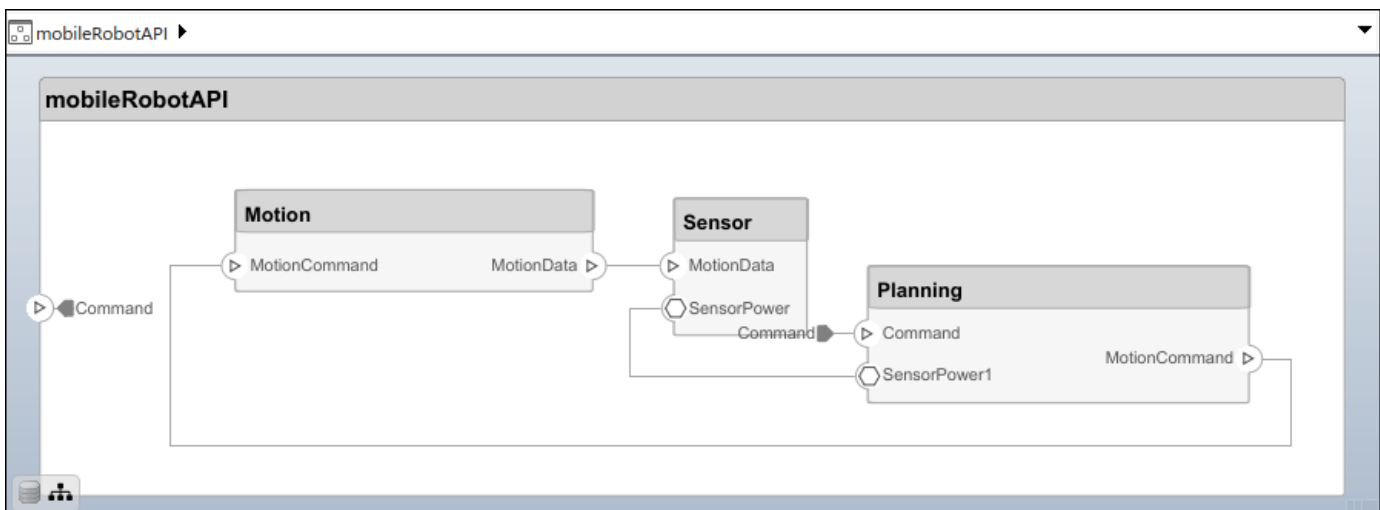
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");  
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID',Type="uint8");  
addProperty(elemSType, 'Description',Type="string");  
addProperty(pCompSType, 'Cost',Type="double",Units="USD");  
addProperty(pCompSType, 'Weight',Type="double",Units="g");  
addProperty(sCompSType, 'develCost',Type="double",Units="USD");  
addProperty(sCompSType, 'develTime',Type="double",Units="hour");  
addProperty(sConnSType, 'unitCost',Type="double",Units="USD");  
addProperty(sConnSType, 'unitWeight',Type="double",Units="g");  
addProperty(sConnSType, 'length',Type="double",Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model,"GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning,"GeneralProfile.softwareComponent")  
applyStereotype(componentSensor,"GeneralProfile.physicalComponent")  
applyStereotype(componentMotion,"GeneralProfile.physicalComponent")
```



Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

## Add Hierarchy

Add two components named Controller and Scope inside the Motion component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);
```

For outport connections, the data element must be specified.

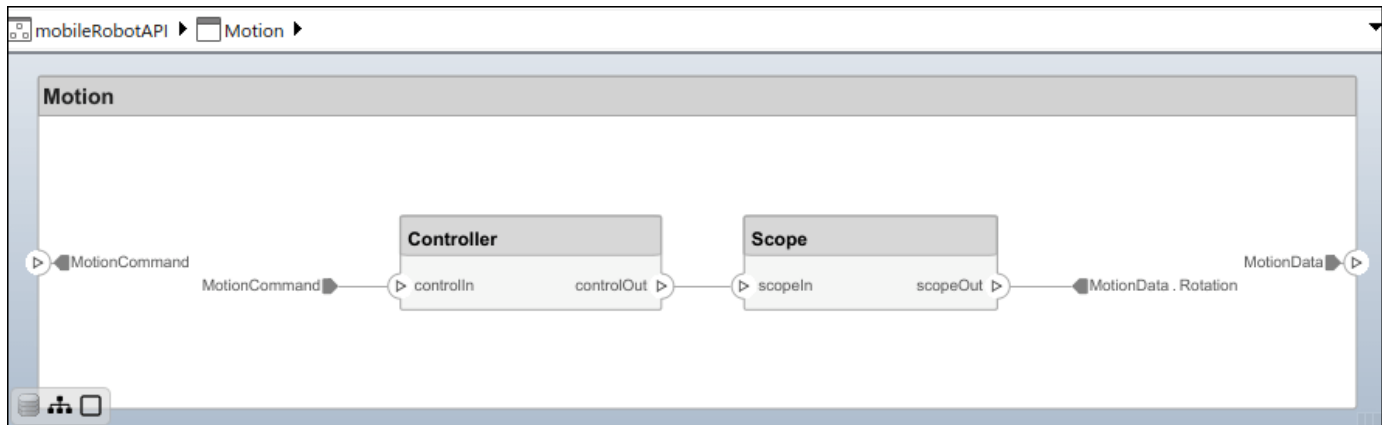
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save
```

```
linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the `Planning` component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

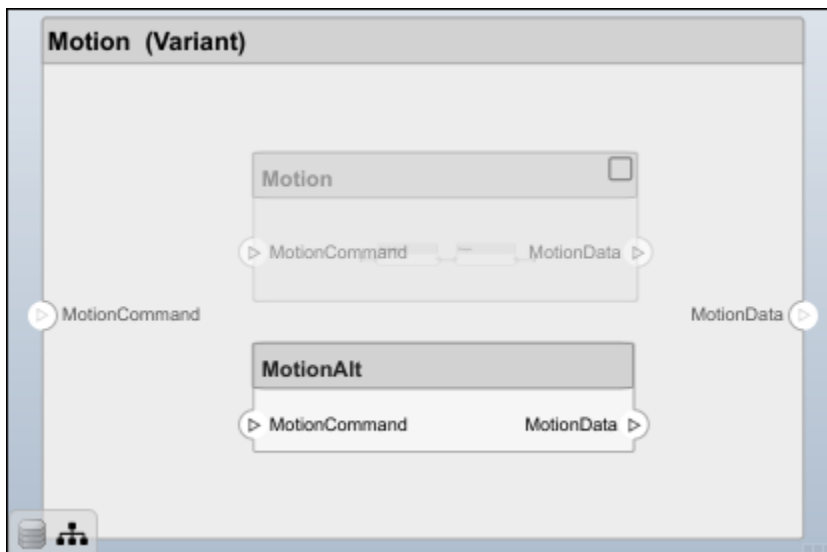
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

## Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>Port interfaces using the <b>Interface Editor</b></li> <li>Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

systemcomposer.arch.Element | systemcomposer.arch.ComponentPort | systemcomposer.arch.BasePort | addPort | Component

### Topics

"Create Architecture Model"

# systemcomposer.arch.BaseComponent

All components in architecture model

## Description

A `BaseComponent` object cannot be constructed. Either create a `systemcomposer.arch.Component` or `systemcomposer.arch.VariantComponent` object. The `systemcomposer.arch.BaseComponent` class is derived from `systemcomposer.arch.Element`.

## Properties

### **Name — Name of component**

character vector

Name of component, specified as a character vector.

Example: 'newComponent'

Data Types: char

### **Architecture — Architecture that defines component structure**

architecture object

Architecture that defines component structure, specified as a `systemcomposer.arch.Architecture` object. For a component that references a different architecture model, this property returns a handle to the root architecture of that model. For variant components, the architecture is that of the active variant.

### **Parent — Architecture that owns component**

architecture object

Architecture that owns component, specified as a `systemcomposer.arch.Architecture` object.

### **Ports — Input and output ports of component**

component port object

Input and output ports of component, specified as a `systemcomposer.arch.ComponentPort` object.

### **Parameters — Parameters of component**

array of parameter objects

Parameters of component, specified as an array of `systemcomposer.arch.Parameter` objects.

### **OwnedArchitecture — Architecture owned by component**

architecture object

Architecture owned by component, specified as a `systemcomposer.arch.Architecture` object.

### **OwnedPorts — Component ports**

array of component port objects

Component ports, specified as an array of `systemcomposer.arch.ComponentPort` objects. For reference components, this property is empty.

### **Position — Position of component on canvas**

vector of coordinates in pixels

Position of component on canvas, specified as a vector of coordinates in pixels: [left top right bottom].

Data Types: double

### **UUID — Universal unique identifier**

character vector

Universal unique identifier for model component, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### **ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the model component and through all operations that preserve the UUID.

Data Types: char

### **Model — Parent model**

model object

Parent System Composer model of component, specified as a `systemcomposer.arch.Model` object.

### **SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: double

### **SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: double

## Object Functions

<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>isProtected</code>	Find if component reference model is protected
<code>isReference</code>	Find if component is referenced to another model
<code>connect</code>	Create architecture model connections
<code>getPort</code>	Get port from component
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>getParameter</code>	Get parameter from architecture or component
<code>getEvaluatedParameterValue</code>	Get evaluated value of parameter from element
<code>getParameterNames</code>	Get parameter names on element
<code>getParameterValue</code>	Get value of parameter
<code>setParameterValue</code>	Set value of parameter
<code>setUnit</code>	Set units on parameter value
<code>resetParameterToDefault</code>	Reset parameter on component to default value
<code>destroy</code>	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
```



```

valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.sldd");

```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```

componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)

```

```

componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)

```

```

componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});

```

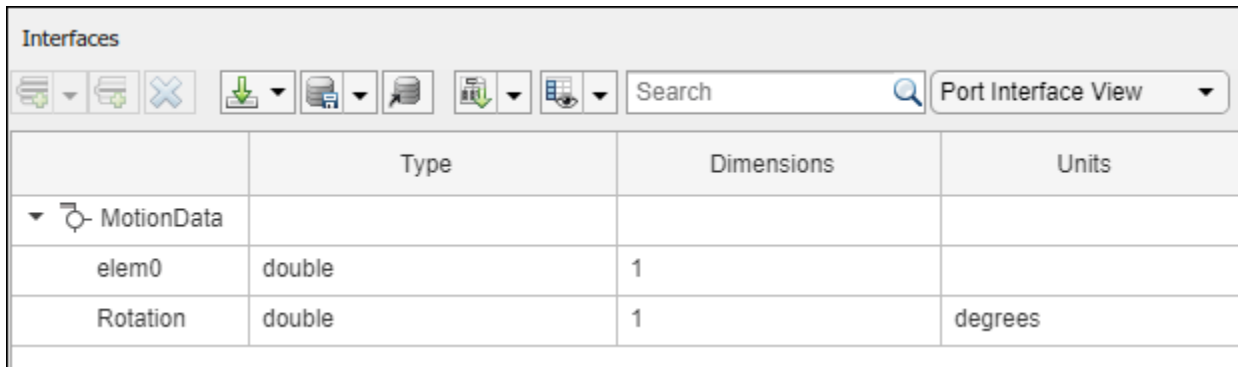
Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```

ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");

```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch,"Command","in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

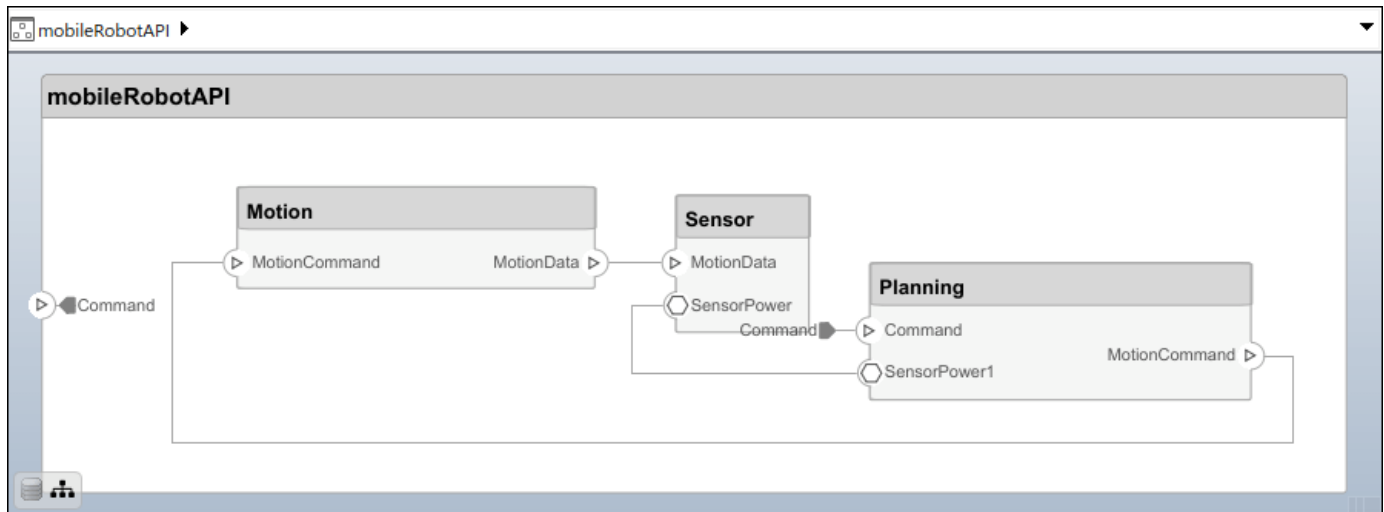
```
compPort = getPort(componentPlanning,"Command");
c_Command = connect(archPort,compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
```

```
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

## Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

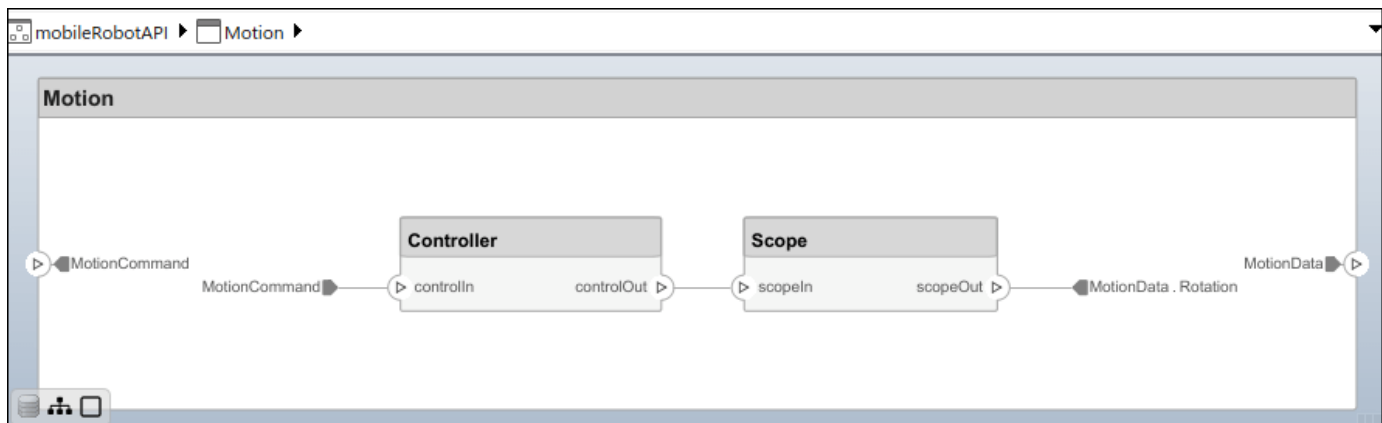
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

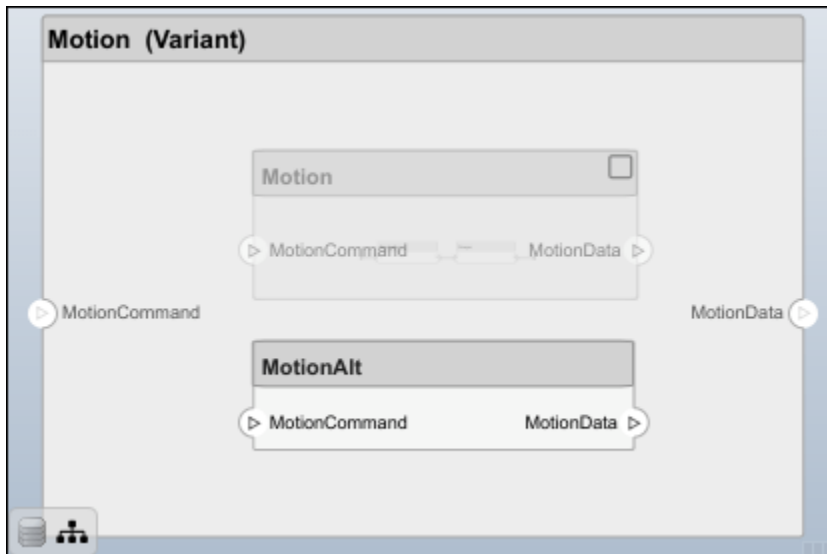
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”



## **Version History**

**Introduced in R2019b**

### **See Also**

Component | systemcomposer.arch.Element | systemcomposer.arch.VariantComponent | systemcomposer.arch.Component

### **Topics**

“Create Architecture Model”

# systemcomposer.arch.BaseConnector

All connectors in architecture model

## Description

A BaseConnector object cannot be constructed. Create either a systemcomposer.arch.Connector or a systemcomposer.arch.PhysicalConnector object. The systemcomposer.arch.BaseConnector class is derived from systemcomposer.arch.Element.

## Properties

### Name — Name of connector

character vector

Name of connector, specified as a character vector.

Example: 'newConnector'

Data Types: char

### Parent — Architecture that owns connector

architecture object

Architecture that owns connector, specified as a systemcomposer.arch.Architecture object.

### Ports — Ports of connection

array of port objects

Ports of connection, specified as an array of systemcomposer.arch.ArchitecturePort or systemcomposer.arch.ComponentPort objects.

### UUID — Universal unique identifier

character vector

Universal unique identifier for model connector, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### ExternalUUID — Unique external identifier

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the model connector and through all operations that preserve the UUID.

Data Types: char

### Model — Parent model

model object

Parent System Composer model of connector, specified as a systemcomposer.arch.Model object.

**SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: double

**SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: double

**Object Functions**

<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>getDestinationElement</code>	Gets data elements selected on destination port for connection
<code>getSourceElement</code>	Gets data elements selected on source port for connection
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>destroy</code>	Remove model element

**Examples****Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

**Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

## Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType", Units="dB", ...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface, "ElectricalElement", ...
    Type="electrical.electrical");
linkDictionary(model, "SensorInterfaces.sidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

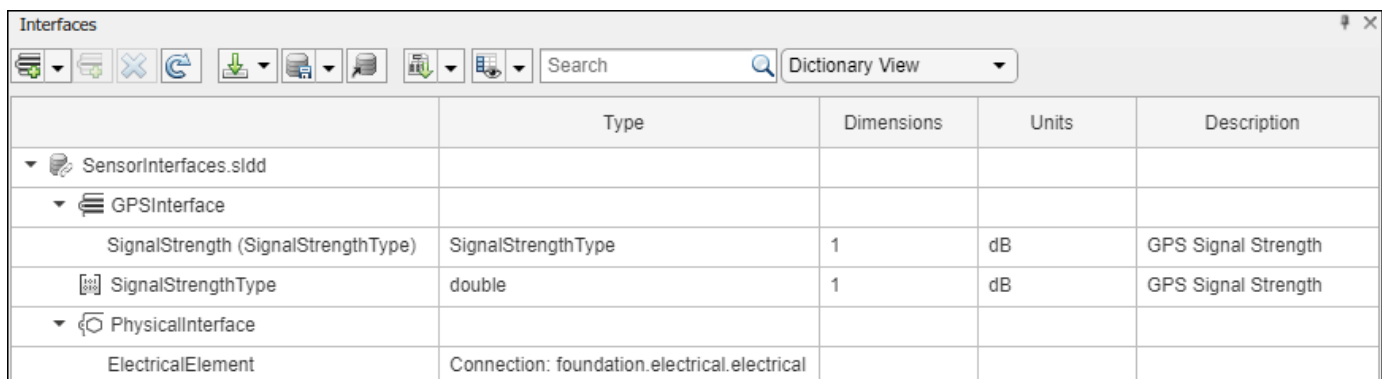
Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.



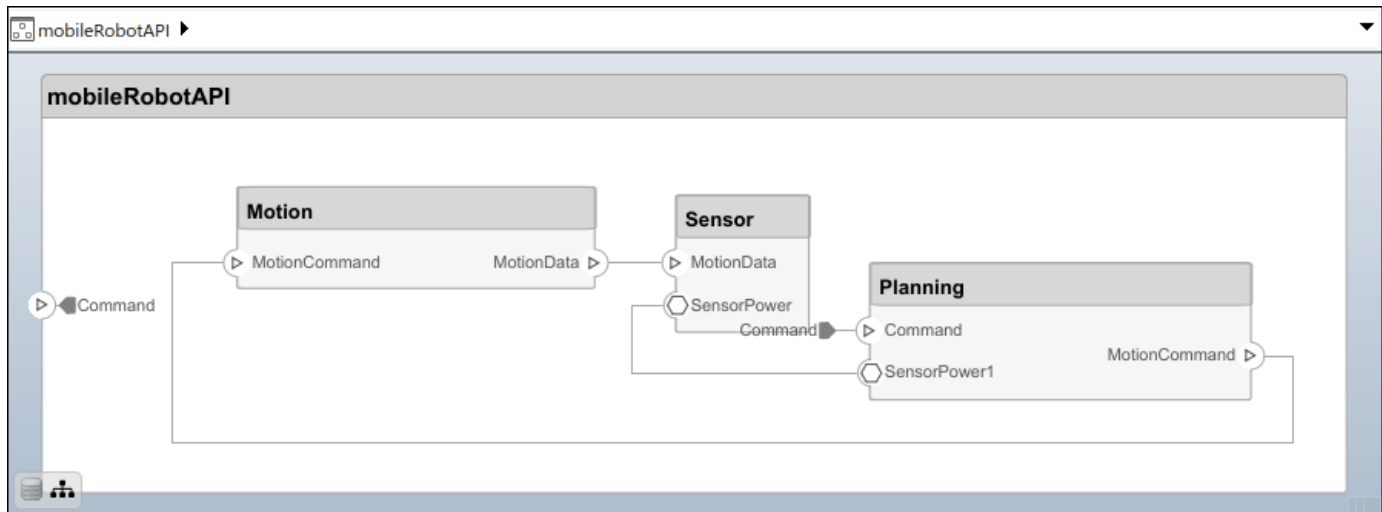
	Type	Dimensions	Units	Description
▼ SensorInterfaces.sidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch, "Sensor");
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, ...
    {'in', 'physical'});
sensorPorts(2).setInterface(physicalInterface)
```



```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

#### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

#### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
```

```

addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");

```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```

applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")

```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```

batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");

```

Set properties for each component.

```

setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

## Add Hierarchy

Add two components named `Controller` and `Scope` inside the `Motion` component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

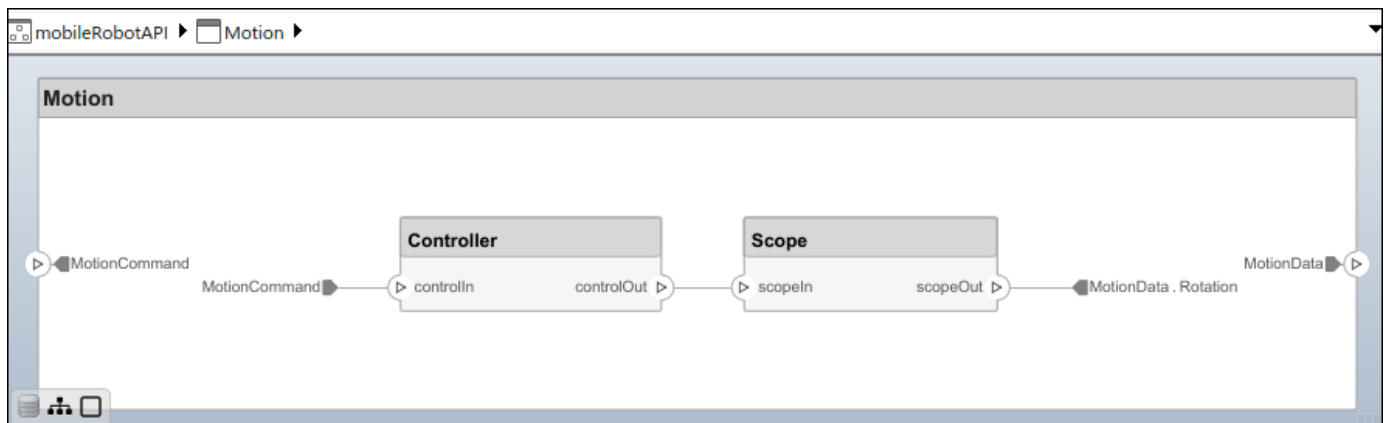
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.



Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

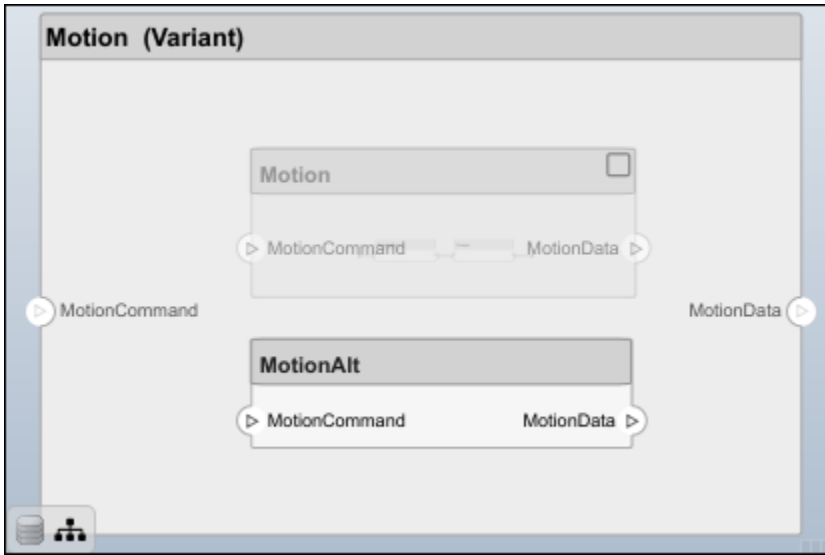
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2021b

### See Also

`systemcomposer.arch.Element` | `systemcomposer.arch.Connector` | `systemcomposer.arch.PhysicalConnector` | `Component`

**Topics**

“Create Architecture Model”

“Implement Component Behavior Using Simscape”

# systemcomposer.arch.BasePort

All ports in architecture model

## Description

A BasePort object cannot be constructed. Create either a systemcomposer.arch.ArchitecturePort or a systemcomposer.arch.ComponentPort object. The systemcomposer.arch.BasePort class is derived from systemcomposer.arch.Element.

## Properties

### Name — Name of port

character vector

Name of port, specified as a character vector.

Example: 'newPort'

Data Types: char

### Direction — Port direction

'Input' | 'Output' | 'Physical' | 'Client' | 'Server'

Port direction, specified as a character vector.

Data Types: char

### Parent — Architecture that owns port

architecture object

Architecture that owns port, specified as a systemcomposer.arch.Architecture object.

### InterfaceName — Name of interface associated with port

character vector

Name of interface associated with port, specified as a character vector.

Data Types: char

### Interface — Interface associated with port

data interface object | value type object | physical interface object | service interface object

Interface associated with port, specified as a systemcomposer.interface.DataInterface, systemcomposer.ValueType, systemcomposer.interface.PhysicalInterface, or systemcomposer.interface.ServiceInterface object.

### Connectors — Port connectors

array of connector objects

Port connectors, specified as an array of systemcomposer.arch.Connector or systemcomposer.arch.PhysicalConnector objects.

**Connected — Whether port has connections**

true or 1 | false or 0

Whether port has connections, specified as a logical.

Data Types: logical

**UUID — Universal unique identifier**

character vector

Universal unique identifier for model port, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the model port and through all operations that preserve the UUID.

Data Types: char

**Model — Parent model**

model object

Parent System Composer model of port, specified as a `systemcomposer.arch.Model` object.**SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: double

**SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: double

**Object Functions**`getProperty`

Get property value corresponding to stereotype applied to element

`setProperty`

Set property value corresponding to stereotype applied to element

`getPropertyValue`

Get value of architecture property

<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>destroy</code>	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.



```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.

	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

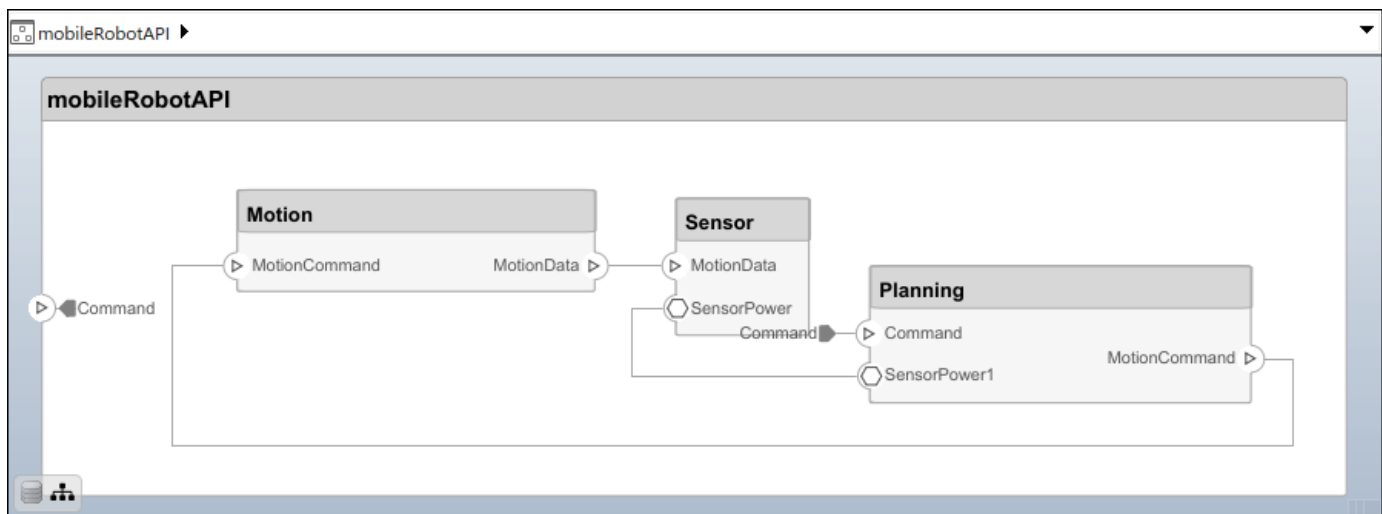
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile, "projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile, "physicalComponent", AppliesTo="Component");
sCompSType = addStereotype(profile, "softwareComponent", AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile, "standardConn", AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
```

```

    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

### Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```

motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);

```

For output connections, the data element must be specified.

```

c_planningScope = connect(scopeCompPortOut, motionPorts(2), DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, ...
    "GeneralProfile.standardConn");

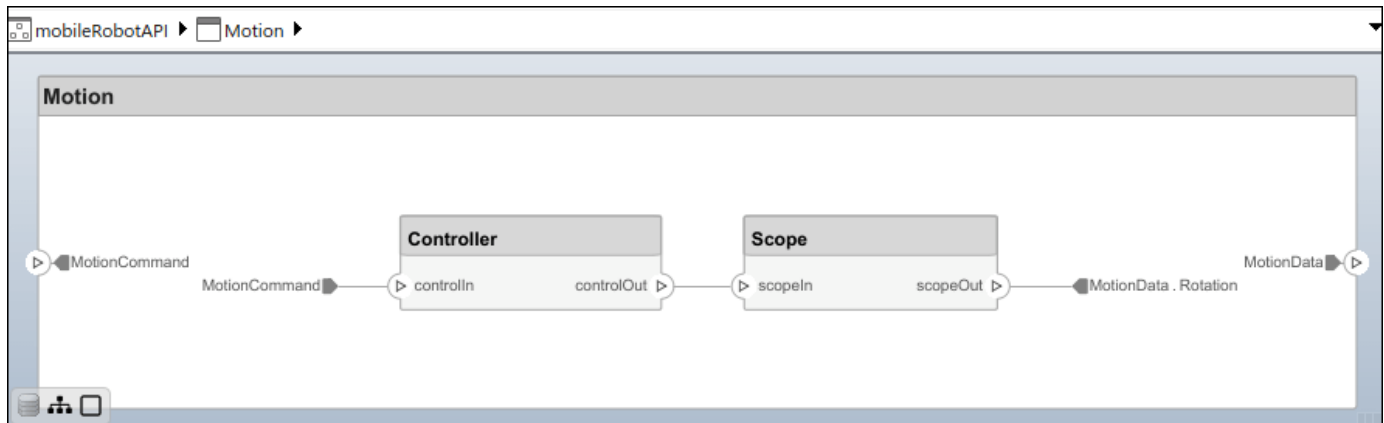
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the **Controller** component into a reference component to reference the new model. To add additional ports on the **Controller** component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "Gyroscope");
referenceModel.save
```

```
linkToModel(motionController, "mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the **Planning** component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active

choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

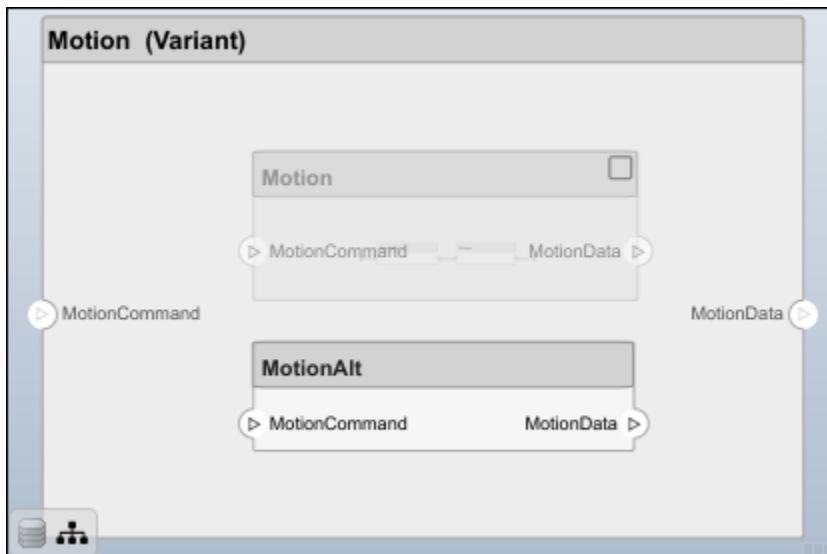
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

cleanUpArtifacts

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	<p>Perform operations on a model:</p> <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> <p>A System Composer model is stored as an SLX file.</p>	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

`systemcomposer.arch.Element` | `systemcomposer.arch.ComponentPort` | `systemcomposer.arch.ArchitecturePort` | `Component`

### Topics

"Create Architecture Model"



# systemcomposer.arch.Component

System Composer component

## Description

A Component object represents a component in a System Composer model. This class inherits from `systemcomposer.arch.BaseComponent`. This class is derived from `systemcomposer.arch.Element`.

## Creation

Create a component in an architecture model using the `addComponent` function.

```
model = systemcomposer.createModel('archModel');
arch = get(model, 'Architecture');
component = addComponent(arch, 'newComponent');
```

## Properties

### Name — Name of component

character vector

Name of component, specified as a character vector.

Example: 'newComponent'

Data Types: char

### Parent — Architecture that owns component

architecture object

Architecture that owns component, specified as a `systemcomposer.arch.Architecture` object.

### Architecture — Architecture that defines component structure

architecture object

Architecture that defines component structure, specified as a `systemcomposer.arch.Architecture` object. For a component that references a different architecture model, this property returns a handle to the root architecture of that model. For variant components, the architecture is that of the active variant.

### OwnedArchitecture — Architecture that component owns

architecture object

Architecture that component owns, specified as a `systemcomposer.arch.Architecture` object. For components that reference an architecture, this property is empty. For variant components, this property is the architecture in which the individual variant components reside.

### Ports — Array of component ports

array of component port objects

Array of component ports, specified as an array of `systemcomposer.arch.ComponentPort` objects.

**OwnedPorts — Array of component ports**

array of component port objects

Array of component ports, specified as an array of `systemcomposer.arch.ComponentPort` objects. For reference components, this property is empty.

**Parameters — Parameters of component**

array of parameter objects

Parameters of component, specified as an array of `systemcomposer.arch.Parameter` objects.

**Position — Position of component on canvas**

vector of coordinates in pixels

Position of component on canvas, specified as a vector of coordinates, in pixels [left top right bottom].

**ReferenceName — Name of model that component references**

character vector

Name of model that component references if linked component, specified as a character vector.

Data Types: char

**IsAdapterComponent — Whether component is adapter block**

true or 1 | false or 0

Whether component is adapter block, specified as a logical.

Data Types: logical

**UUID — Universal unique identifier**

character vector

Universal unique identifier for model component, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the model component and through all operations that preserve the UUID.

Data Types: char

**Model — Parent model**

model object

Parent System Composer model of component, specified as a `systemcomposer.arch.Model` object.

**SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: `double`

### **SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: `double`

## **Object Functions**

<code>createArchitectureModel</code>	Create architecture model from component
<code>createSimulinkBehavior</code>	Create Simulink behavior and link to component
<code>createStateflowChartBehavior</code>	Add Stateflow chart behavior to component
<code>linkToModel</code>	Link component to model
<code>inlineComponent</code>	Remove reference architecture or behavior from component
<code>makeVariant</code>	Convert component to variant choice
<code>isProtected</code>	Find if component reference model is protected
<code>isReference</code>	Find if component is referenced to another model
<code>connect</code>	Create architecture model connections
<code>getPort</code>	Get port from component
<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>getQualifiedName</code>	Get model element qualified name
<code>getParameter</code>	Get parameter from architecture or component
<code>getEvaluatedParameterValue</code>	Get evaluated value of parameter from element
<code>getParameterNames</code>	Get parameter names on element
<code>getParameterValue</code>	Get value of parameter
<code>setParameterValue</code>	Set value of parameter
<code>setUnit</code>	Set units on parameter value
<code>resetParameterToDefault</code>	Reset parameter on component to default value
<code>destroy</code>	Remove model element

## **Examples**

## Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");  
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");  
interface = dictionary.addInterface("GPSInterface");  
element = interface.addElement("SignalStrength");  
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...  
    Description="GPS Signal Strength");  
element.setType(valueType);  
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");  
physicalElement = addElement(physicalInterface,"ElectricalElement",...  
    Type="electrical.electrical");  
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
SensorInterfaces.sidd				
GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
  {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
  {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
  {'in','out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.

	Type	Dimensions	Units
MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

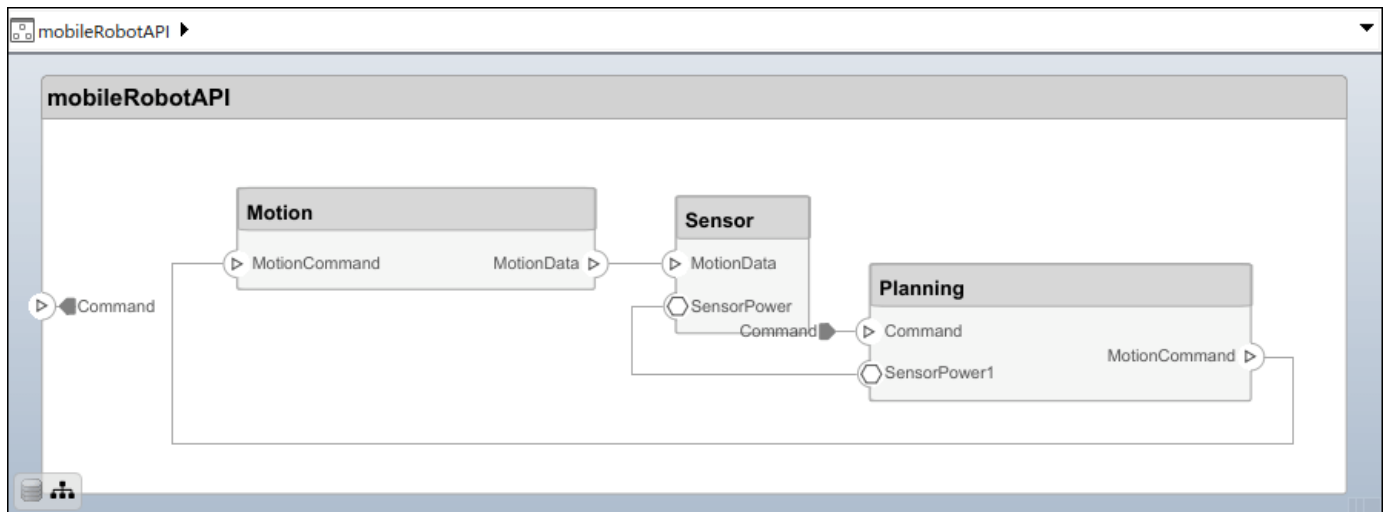
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

#### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile, "projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile, "physicalComponent", AppliesTo="Component");
sCompSType = addStereotype(profile, "softwareComponent", AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile, "standardConn", AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
```

```

setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

### Add Hierarchy

Add two components named Controller and Scope inside the Motion component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```

motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);

```

For output connections, the data element must be specified.

```

c_planningScope = connect(scopeCompPortOut, motionPorts(2), DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, ...
    "GeneralProfile.standardConn");

```

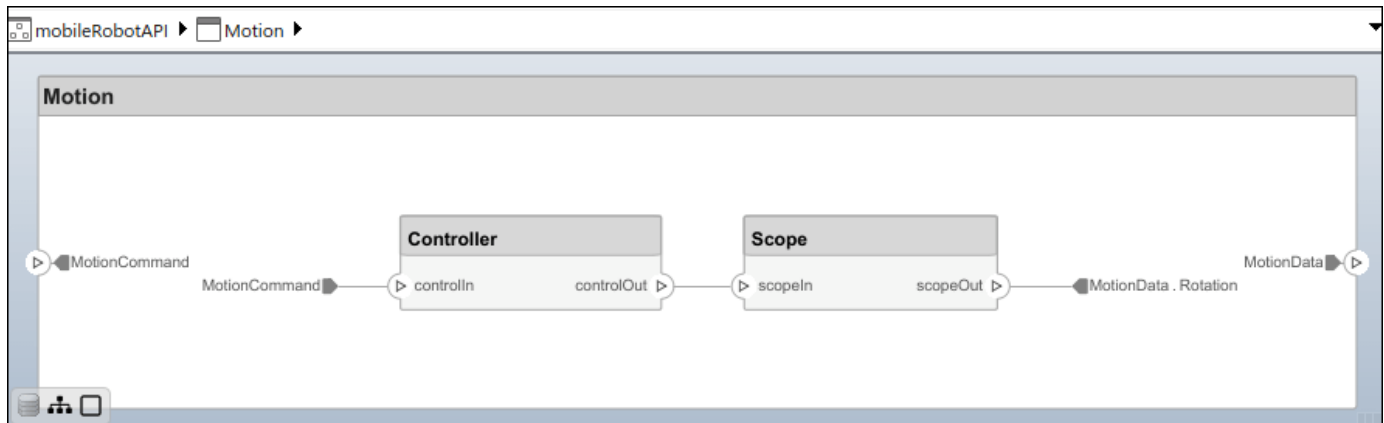
Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```





### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "Gyroscope");
referenceModel.save
```

```
linkToModel(motionController, "mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the makeVariant function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active

choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

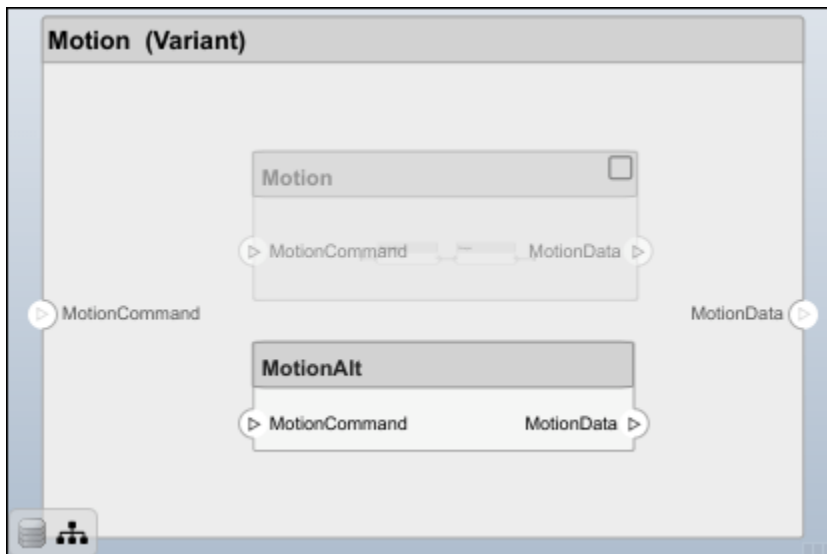
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

cleanUpArtifacts

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

`systemcomposer.arch.Architecture` | `systemcomposer.arch.Element` | `createModel` | `addComponent` | `Component`

### Topics

"Create Architecture Model"

# systemcomposer.arch.ComponentPort

Component port

## Description

A ComponentPort object represents the input, output, and physical ports of a System Composer component. This class inherits from `systemcomposer.arch.BasePort`. This class is derived from `systemcomposer.arch.Element`.

## Creation

A component port is constructed by creating an architecture port on the architecture of the component using the `addPort` function, then getting the component port using the `getPort` function.

```
addPort(compObj.Architecture, 'portName', 'in');  
compPortObj = getPort(compObj, 'portName');
```

## Properties

### Name — Name of port

character vector

Name of port, specified as a character vector.

Example: 'portName'

Data Types: char

### Direction — Port direction

'Input' | 'Output' | 'Physical' | 'Client' | 'Server'

Port direction, specified as a character vector.

Data Types: char

### InterfaceName — Name of interface

character vector

Name of interface associated with port, specified as a character vector.

Data Types: char

### Interface — Interface associated with port

data interface object | value type object | physical interface object | service interface object

Interface associated with port, specified as a `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

**Connectors — Port connectors**

array of connector objects

Port connectors, specified as an array of `systemcomposer.arch.Connector` or `systemcomposer.arch.PhysicalConnector` objects.

**Connected — Whether port has connections**

true or 1 | false or 0

Whether port has connections, specified as a logical.

Data Types: `logical`

**Parent — Component that owns port**

architecture object

Component that owns port, specified as a `systemcomposer.arch.Architecture` object.

**ArchitecturePort — Architecture port**

architecture port object

Architecture port within the component that maps to port, specified as a `systemcomposer.arch.ArchitecturePort` object.

**UUID — Universal unique identifier**

character vector

Universal unique identifier for model component port, specified as a character vector.

Example: `'91d5de2c-b14c-4c76-a5d6-5dd0037c52df'`

Data Types: `char`

**ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the model component port and through all operations that preserve the UUID.

Data Types: `char`

**Model — Parent model**

model object

Parent System Composer model of port, specified as a `systemcomposer.arch.Model` object.

**SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: `double`

**SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a **double**.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: `double`

**Object Functions**

<code>setName</code>	Set name for port
<code>setInterface</code>	Set interface for port
<code>createInterface</code>	Create and set owned interface for port
<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>connect</code>	Create architecture model connections
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>getQualifiedname</code>	Get model element qualified name

**Examples****Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

**Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

**Add Components, Ports, Connections, and Interfaces**

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```

dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.slidd");

```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.slidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```

componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)

```

```

componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)

```

```

componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});

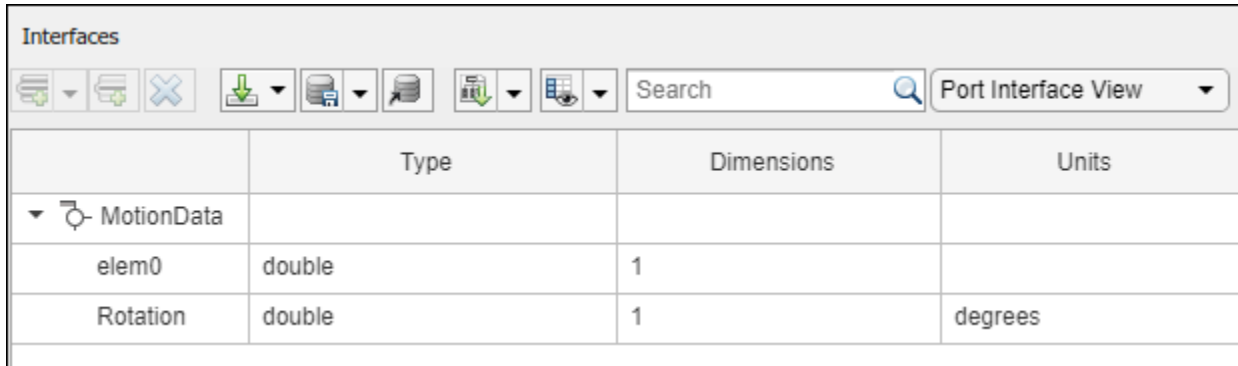
```



Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.



```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces			
 <input type="text" value="Search"/> <input type="button" value="Port Interface View"/>			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch,"Command","in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

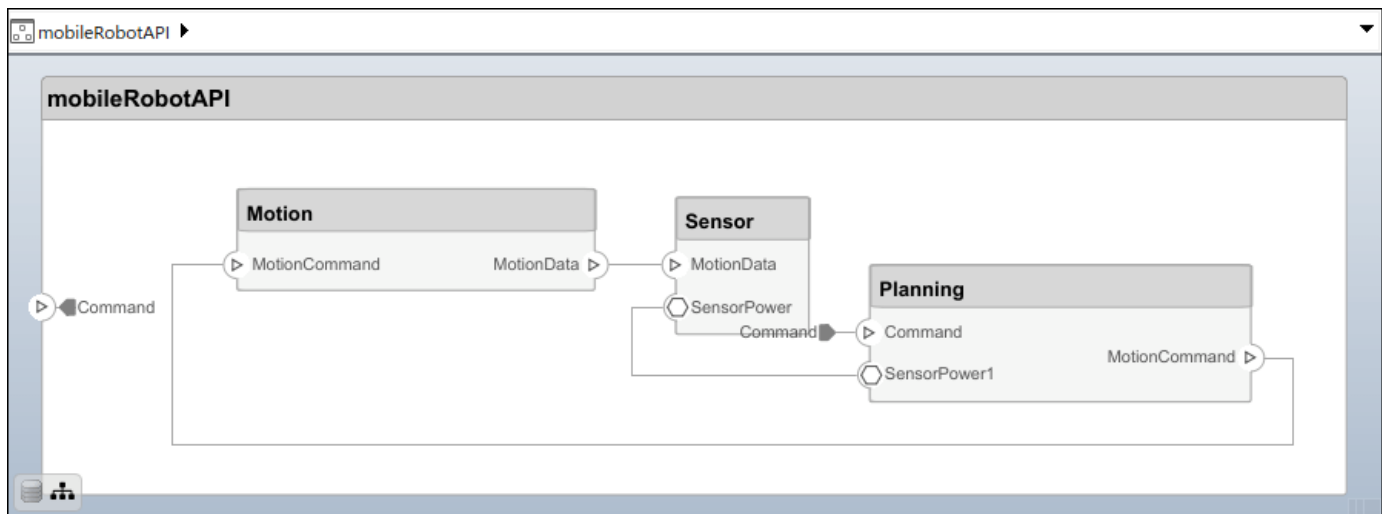
```
compPort = getPort(componentPlanning,"Command");
c_Command = connect(archPort,compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

#### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

#### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
```

```

addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");

```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```

applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")

```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```

batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");

```

Set properties for each component.

```

setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

## Add Hierarchy

Add two components named `Controller` and `Scope` inside the `Motion` component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

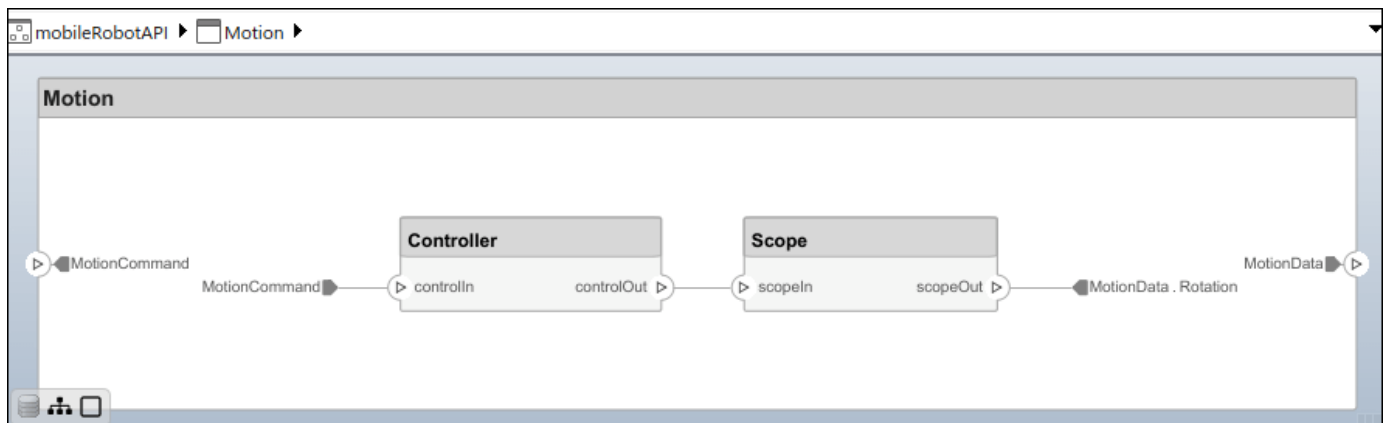
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "Gyroscope");
referenceModel.save

linkToModel(motionController, "mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

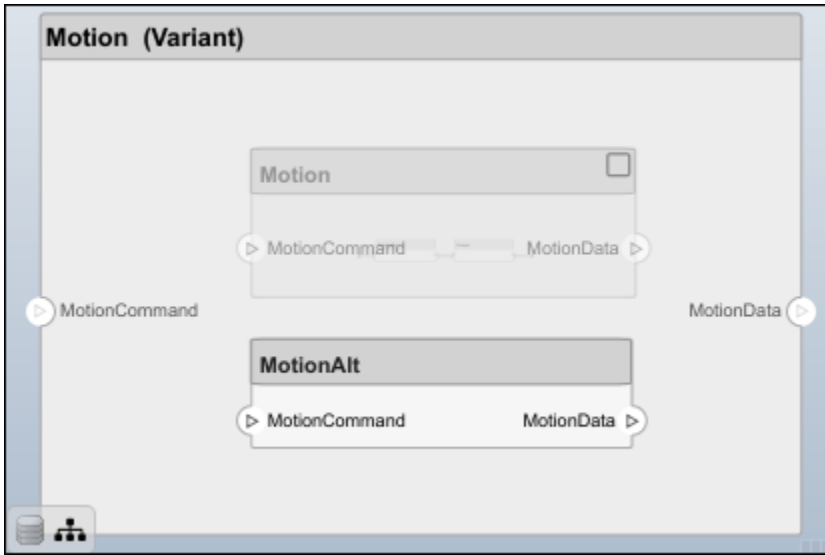
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## **Version History**

**Introduced in R2019a**

### **See Also**

systemcomposer.arch.ArchitecturePort | systemcomposer.arch.BasePort |  
systemcomposer.arch.Element | getPort | addPort | Component

### **Topics**

“Create Architecture Model”



# systemcomposer.arch.Connector

Connector between ports

## Description

A Connector object represents a connector between ports for a System Composer model. This class inherits from `systemcomposer.arch.BaseConnector`. This class is derived from `systemcomposer.arch.Element`.

## Creation

Create connectors using the `connect` function.

```
conns = connect(architecture,outPorts,inPorts)
```

## Properties

### Parent — Parent architecture that owns connector

architecture object

Parent architecture that owns connector, specified as a `systemcomposer.arch.Architecture` object.

### Name — Name of connector

character vector

Name of connector, specified as a character vector.

Data Types: `char`

### SourcePort — Source of connection

architecture port object | component port object

Source of connection as output port, specified as a `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` object.

### DestinationPort — Destination of connection

architecture port object | component port object

Destination of connection as input port, specified as a `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` object.

### Ports — Ports of connection

array of port objects

Ports of connection, specified as an array of `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` objects.

### UUID — Universal unique identifier

character vector

Universal unique identifier for model connector, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### **ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the model connector and through all operations that preserve the UUID.

Data Types: char

### **Model — Parent model**

model object

Parent System Composer model of connector, specified as a `systemcomposer.arch.Model` object.

### **SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: `double`

### **SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: `double`

## **Object Functions**

<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>getSourceElement</code>	Gets data elements selected on source port for connection
<code>getDestinationElement</code>	Gets data elements selected on destination port for connection
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property

getQualifiedName	Get model element qualified name
destroy	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType", Units="dB", ...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface, "ElectricalElement", ...
    Type="electrical.electrical");
linkDictionary(model, "SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

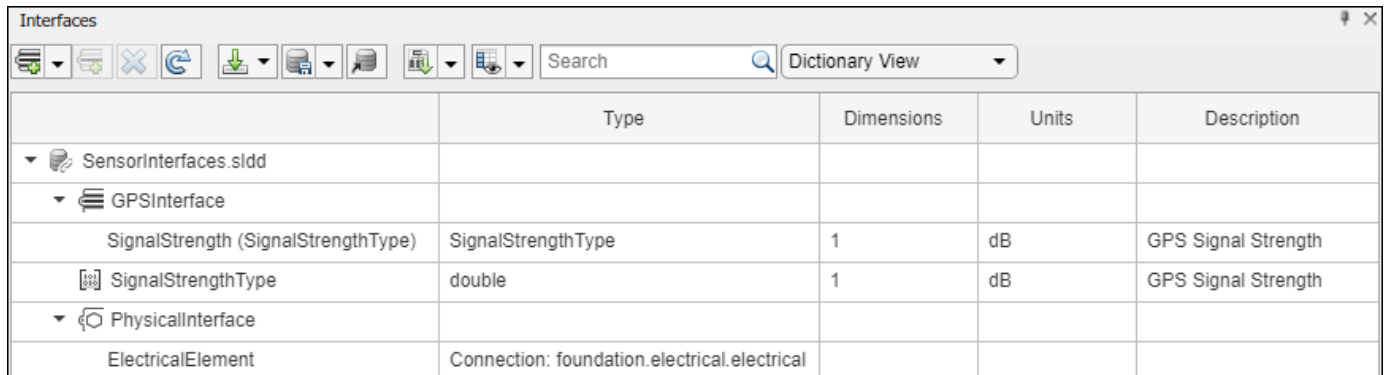
Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.



	Type	Dimensions	Units	Description
▼ SensorInterfaces.sidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

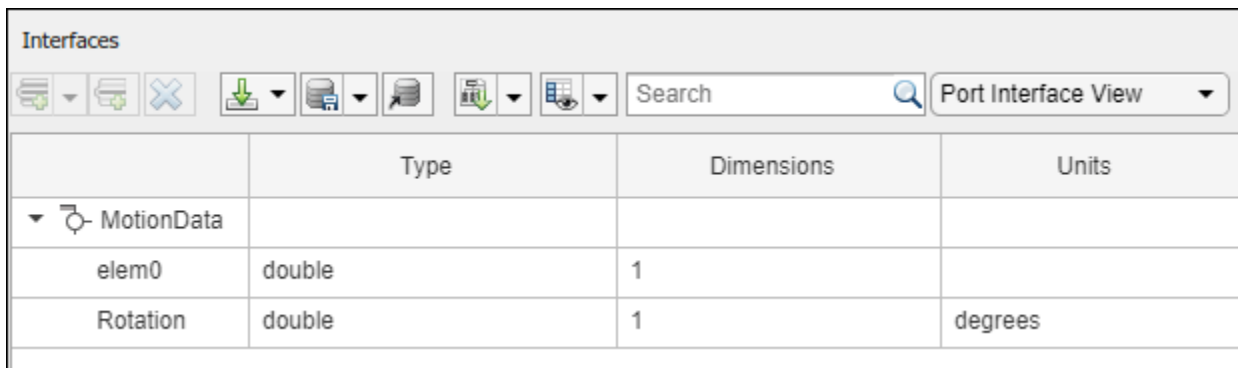
```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

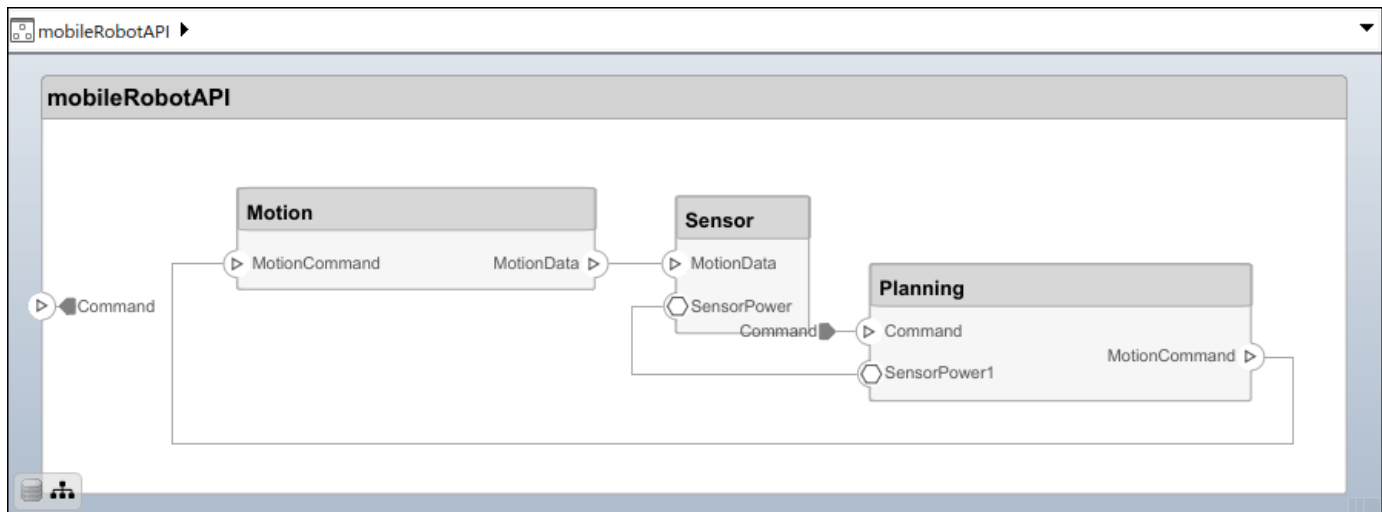
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

#### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile, "projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile, "physicalComponent", AppliesTo="Component");
sCompSType = addStereotype(profile, "softwareComponent", AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile, "standardConn", AppliesTo="Connector");
```

### **Add Properties**

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### **Save Profile**

```
profile.save;
```

### **Apply Profile to Model**

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
```

```

setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

## Add Hierarchy

Add two components named Controller and Scope inside the Motion component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```

motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);

```

For output connections, the data element must be specified.

```

c_planningScope = connect(scopeCompPortOut, motionPorts(2), DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, ...
    "GeneralProfile.standardConn");

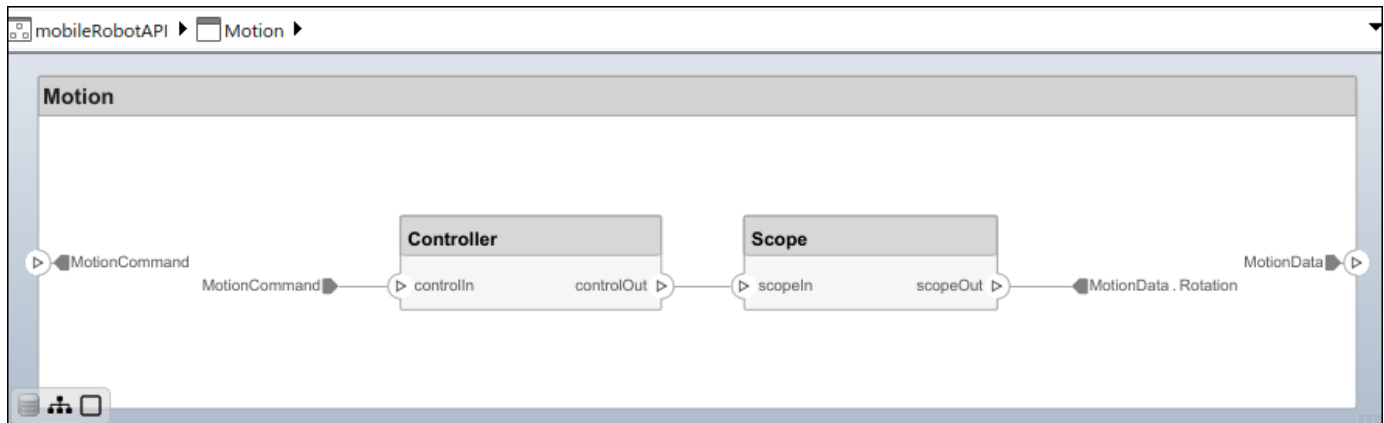
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the **Controller** component into a reference component to reference the new model. To add additional ports on the **Controller** component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "Gyroscope");
referenceModel.save
```

```
linkToModel(motionController, "mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the **Planning** component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active



choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

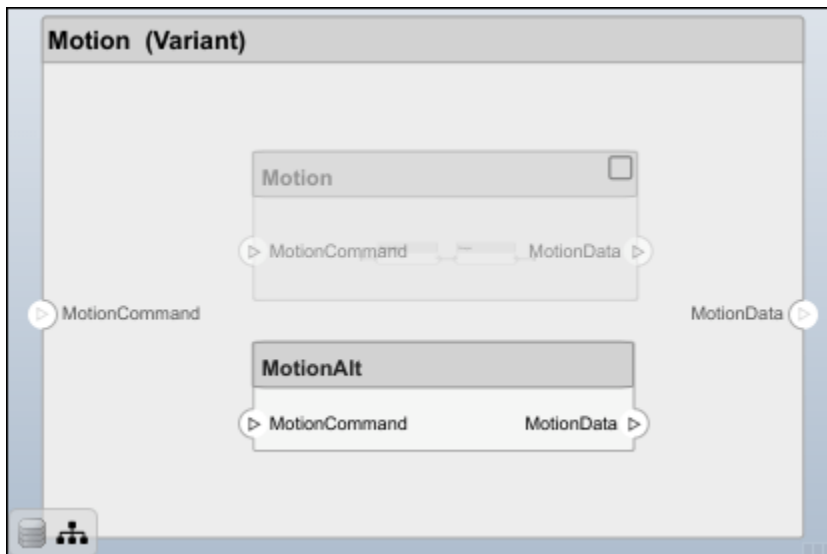
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

cleanUpArtifacts

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	<p>Perform operations on a model:</p> <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> <p>A System Composer model is stored as an SLX file.</p>	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## Version History

Introduced in R2019a

### See Also

systemcomposer.arch.Element | systemcomposer.arch.BaseConnector | systemcomposer.arch.PhysicalConnector | connect | Component

### Topics

“Create Architecture Model”

# systemcomposer.arch.Element

All model elements

## Description

The Element class is the base class for all System Composer model elements:

- `systemcomposer.arch.Architecture`
- `systemcomposer.arch.Component`
- `systemcomposer.arch.VariantComponent`
- `systemcomposer.arch.BaseComponent`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.BasePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.arch.PhysicalConnector`
- `systemcomposer.arch.BaseConnector`

## Creation

Create a component using the `addComponent` function, a port using the `addPort` function, or a connector using the `connect` function.

## Properties

### UUID — Universal unique identifier

character vector

Universal unique identifier for model element, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### ExternalUID — Unique external identifier

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the model element and through all operations that preserve the UUID.

Data Types: char

### Model — Parent model

model object

Parent System Composer model of element, specified as a `systemcomposer.arch.Model` object.

**SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: `double`

**SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: `double`

**Object Functions**

<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>destroy</code>	Remove model element

**Examples****Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

**Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

**Add Components, Ports, Connections, and Interfaces**

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
SensorInterfaces.slidd				
GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'}
    {'in','physical','out'});
```

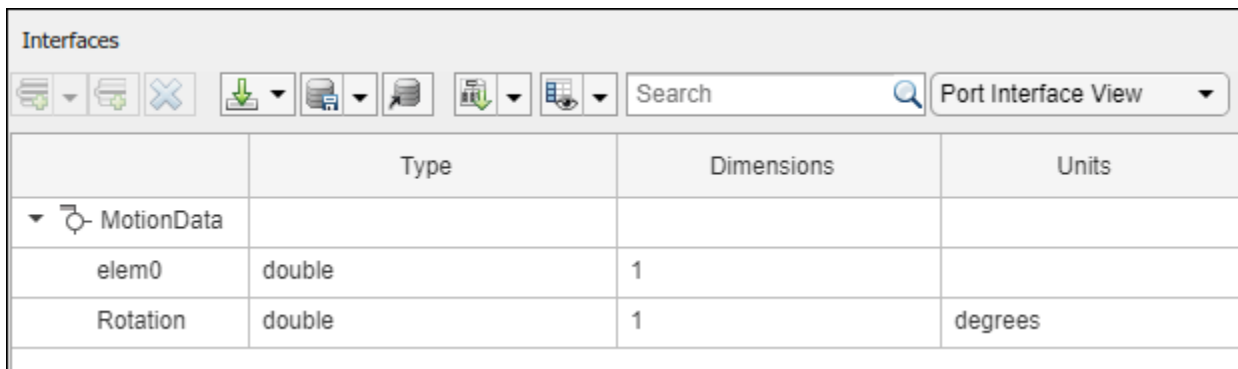
```
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, ...
    {'in', 'out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces			
<span>Icons</span> <input type="text" value="Search"/> <span>Port Interface View</span>			
	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch, componentSensor, componentPlanning, Rule="interface");
c_motionData = connect(arch, componentMotion, componentSensor);
c_motionCommand = connect(arch, componentPlanning, componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

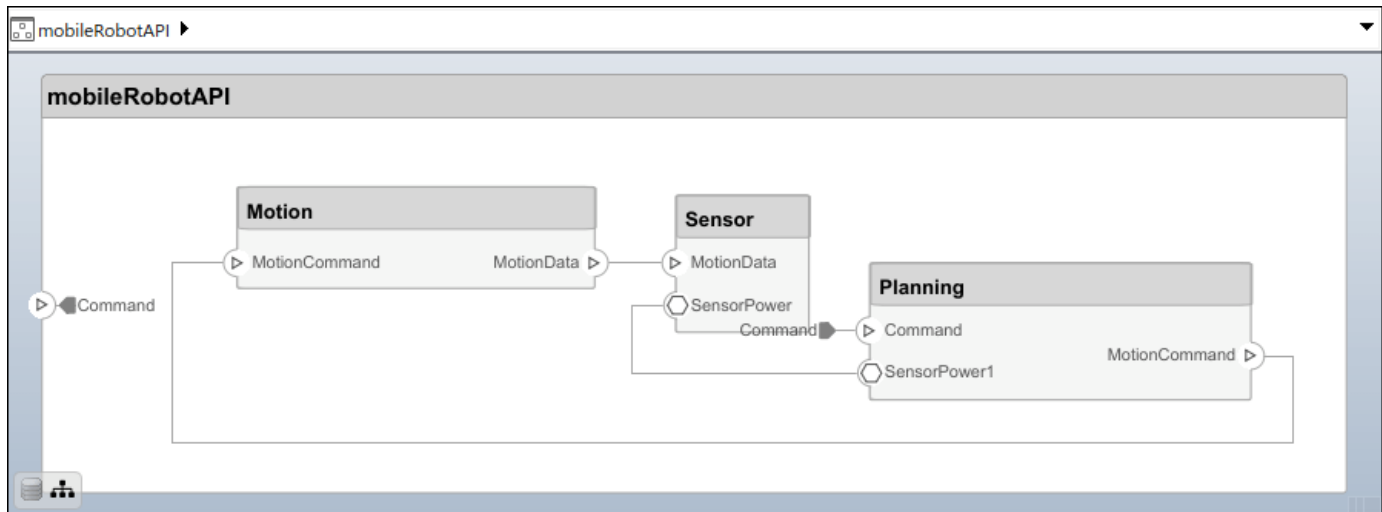
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

#### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

#### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType,'ID',Type="uint8");
addProperty(elemSType,'Description',Type="string");
addProperty(pCompSType,'Cost',Type="double",Units="USD");
addProperty(pCompSType,'Weight',Type="double",Units="g");
addProperty(sCompSType,'develCost',Type="double",Units="USD");
addProperty(sCompSType,'develTime',Type="double",Units="hour");
```



```

addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");

```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```

applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")

```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```

batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");

```

Set properties for each component.

```

setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

## Add Hierarchy

Add two components named `Controller` and `Scope` inside the `Motion` component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

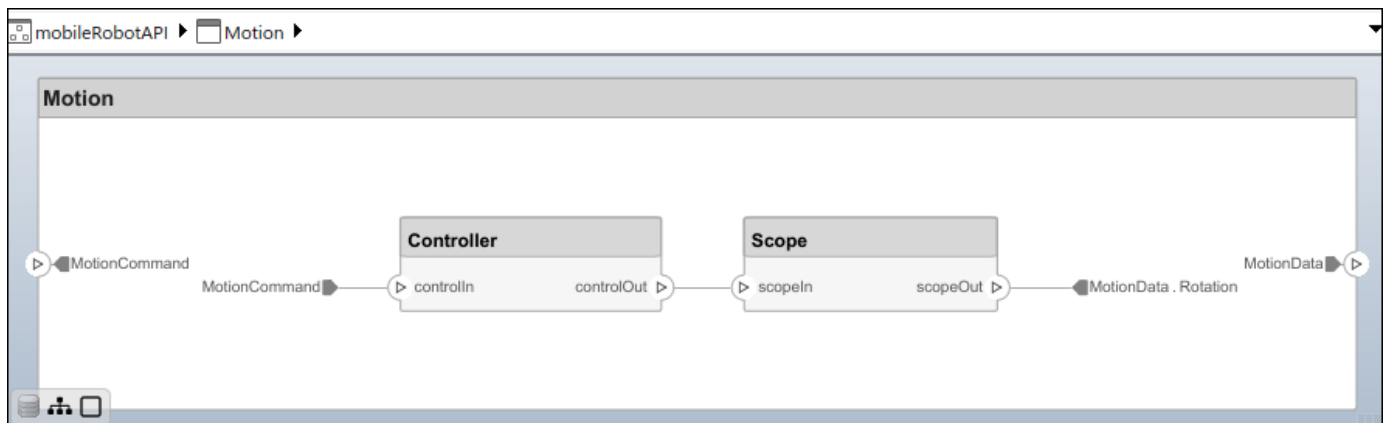
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "Gyroscope");
referenceModel.save

linkToModel(motionController, "mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

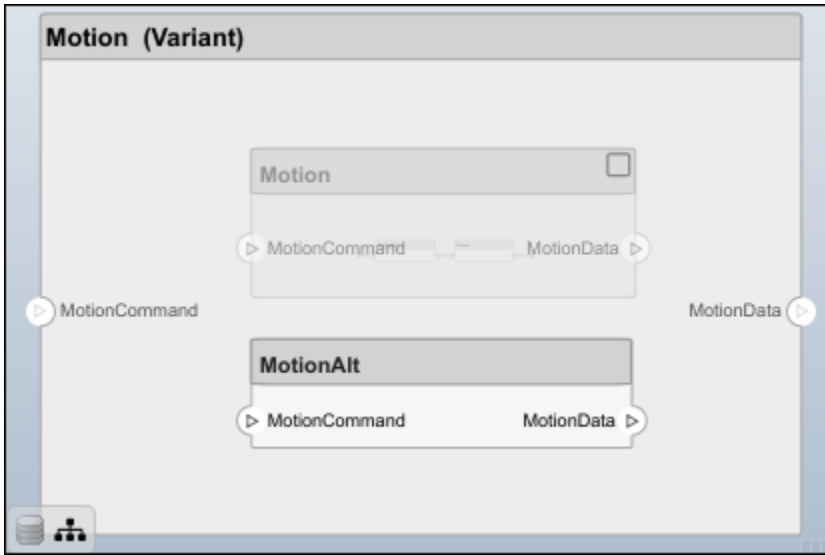
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## **Version History**

**Introduced in R2019a**

### **See Also**

#### **Topics**

“Create Architecture Model”

# systemcomposer.arch.Function

Software architecture function

## Description

A Function object represents a function in a software architecture model.

Use the **Functions Editor** from the toolstrip on a software architecture model, to edit the simulation execution order and sample time of functions with inherited sample time (-1) in your software architecture.

## Creation

Get functions in a software architecture model with the Functions property on the `systemcomposer.arch.Architecture` object.

```
model = systemcomposer.openModel('ThrottleControlComposition');  
sim('ThrottleControlComposition');  
functions = model.Architecture.Functions
```

## Properties

### Model — Architecture model

model object

Architecture model where element belongs, specified as a `systemcomposer.arch.Model` object.

### Name — Name of function

character vector

Name of function, specified as a character vector.

Data Types: char

### Component — Component where function is defined

component object

Component where function is defined, specified as a `systemcomposer.arch.Component` object.

### Parent — Parent architecture of element

architecture object

Parent architecture of element where function is defined, specified as a `systemcomposer.arch.Architecture` object.

### Period — Period of function

numeric | string

Period of function, specified as a numeric value convertible to a string, or a string of valid MATLAB variables. The `Period` property of aperiodic functions is editable. Editing the `Period` property of a periodic function will result in an error.

### **ExecutionOrder — Execution order of functions**

row vector of numeric values

Execution order of functions, specified as a row vector of numeric values.

Example: `[model.Architecture.Functions.ExecutionOrder]`

Data Types: `uint64`

### **UUID — Universal unique identifier**

character vector

Universal unique identifier for function, specified as a character vector.

Example: `'91d5de2c-b14c-4c76-a5d6-5dd0037c52df'`

Data Types: `char`

### **ExternalUUID — Unique external identifier**

character vector

Unique external identifier for function, specified as a character vector. The external ID is preserved over the lifespan of the function and through all operations that preserve the `UUID`.

Data Types: `char`

## **Object Functions**

<code>increaseExecutionOrder</code>	Change function execution order to later
<code>decreaseExecutionOrder</code>	Change function execution order to earlier
<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>destroy</code>	Remove model element

## **Examples**

### **Change Execution Order of Software Functions**

This example shows the software architecture of a throttle position control system and how to schedule the execution order of the root level functions.

```
model = systemcomposer.openModel("ThrottleControlComposition");
```

Simulate the model to populate it with functions.



```
sim("ThrottleControlComposition");
```

View the function names ordered by execution order.

```
functions = {model.Architecture.Functions.Name}'
```

```
functions = 6x1 cell
    {'Actuator_output_5ms' }
    {'Controller_run_5ms' }
    {'TPS_Primary_read_5ms' }
    {'TPS_Secondary_read_5ms' }
    {'TP_Monitor_D1' }
    {'APP_Sensor_read_10ms' }
```

Decrease the execution order of the third function.

```
decreaseExecutionOrder(model.Architecture.Functions(3))
```

View the function names ordered by execution order.

```
functions = {model.Architecture.Functions.Name}'
```

```
functions = 6x1 cell
    {'Actuator_output_5ms' }
    {'TPS_Primary_read_5ms' }
    {'Controller_run_5ms' }
    {'TPS_Secondary_read_5ms' }
    {'TP_Monitor_D1' }
    {'APP_Sensor_read_10ms' }
```

The third function is now moved up in execution order, executing earlier.

Increase the execution order of the second function.

```
increaseExecutionOrder(model.Architecture.Functions(2))
```

View the function names ordered by execution order.

```
functions = {model.Architecture.Functions.Name}'
```

```
functions = 6x1 cell
    {'Actuator_output_5ms' }
    {'Controller_run_5ms' }
    {'TPS_Primary_read_5ms' }
    {'TPS_Secondary_read_5ms' }
    {'TP_Monitor_D1' }
    {'APP_Sensor_read_10ms' }
```

The second function is now moved down in execution order, executing later.

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>• “Author Software Architectures”</li> <li>• “Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	"Class Diagram View of Software Architectures"

## **Version History**

**Introduced in R2021b**

### **See Also**

`systemcomposer.createModel` | `createArchitectureModel` | `createSimulinkBehavior`

### **Topics**

“Modeling Software Architecture of Throttle Position Control System”

“Simulate and Deploy Software Architectures”

“Author Software Architectures”

# systemcomposer.arch.Model

System Composer model

## Description

A Model object is used to manage architecture objects in a System Composer model.

## Creation

Create a model using the createModel function.

```
objModel = systemcomposer.createModel('NewModel')
```

## Properties

### Name — Name of model

character vector

Name of model, specified as a character vector. This property must be a valid MATLAB identifier.

Example: 'NewModel'

Data Types: char

### Architecture — Root architecture

architecture object

Root architecture of model, specified as a systemcomposer.arch.Architecture object.

### SimulinkHandle — Simulink handle

numeric value

Simulink handle, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: handle = get(object, 'SimulinkHandle')

Data Types: double

### Profiles — Profiles

array of profile objects

Profiles attached to the model, specified as an array of systemcomposer.profile.Profile objects.

### InterfaceDictionary — Dictionary object that holds interfaces

dictionary object

Dictionary object that holds interfaces, specified as a `systemcomposer.interface.Dictionary` object. If the model is not linked to an external dictionary, this property is a handle to the implicit dictionary.

### **Views – Views**

array of view objects

Views, specified as an array of `systemcomposer.view.View` objects.

Example: `objView = get(objModel, 'Views')`

## **Object Functions**

<code>open</code>	Open architecture model
<code>close</code>	Close architecture model
<code>save</code>	Save architecture model or data dictionary
<code>find</code>	Find architecture model elements using query
<code>lookup</code>	Search for architectural element
<code>openViews</code>	Open Architecture Views Gallery
<code>createView</code>	Create architecture view
<code>getView</code>	Find architecture view
<code>deleteView</code>	Delete architecture view
<code>applyProfile</code>	Apply profile to model
<code>removeProfile</code>	Remove profile from model
<code>saveToDictionary</code>	Save interfaces to dictionary
<code>linkDictionary</code>	Link data dictionary to architecture model
<code>unlinkDictionary</code>	Unlink data dictionary from architecture model
<code>renameProfile</code>	Rename profile in model
<code>iterate</code>	Iterate over model elements

## **Examples**

### **Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

#### **Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### **Add Components, Ports, Connections, and Interfaces**

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");  
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the

value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
SensorInterfaces.sldd				
GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

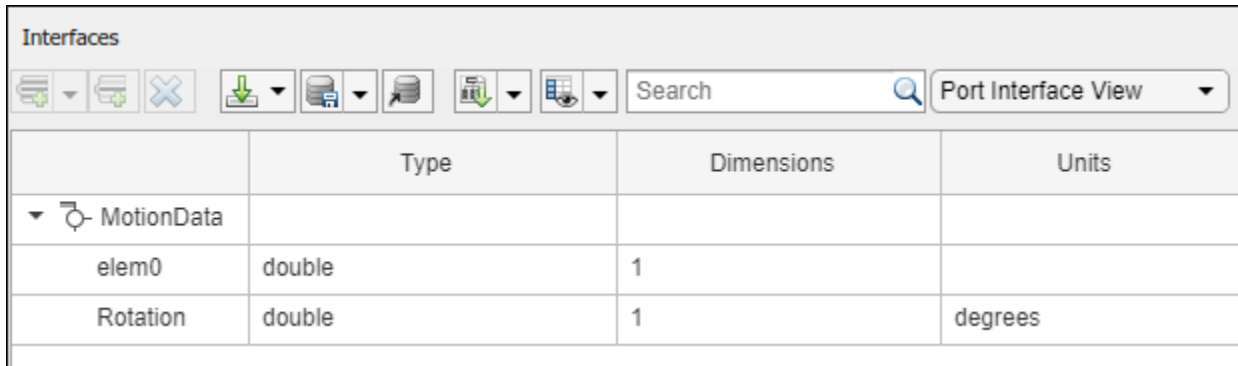
```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)
```



```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces			
 <input type="text" value="Search"/> <span>Port Interface View</span>			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch,"Command","in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

```
compPort = getPort(componentPlanning,"Command");
c_Command = connect(archPort,compPort);
```

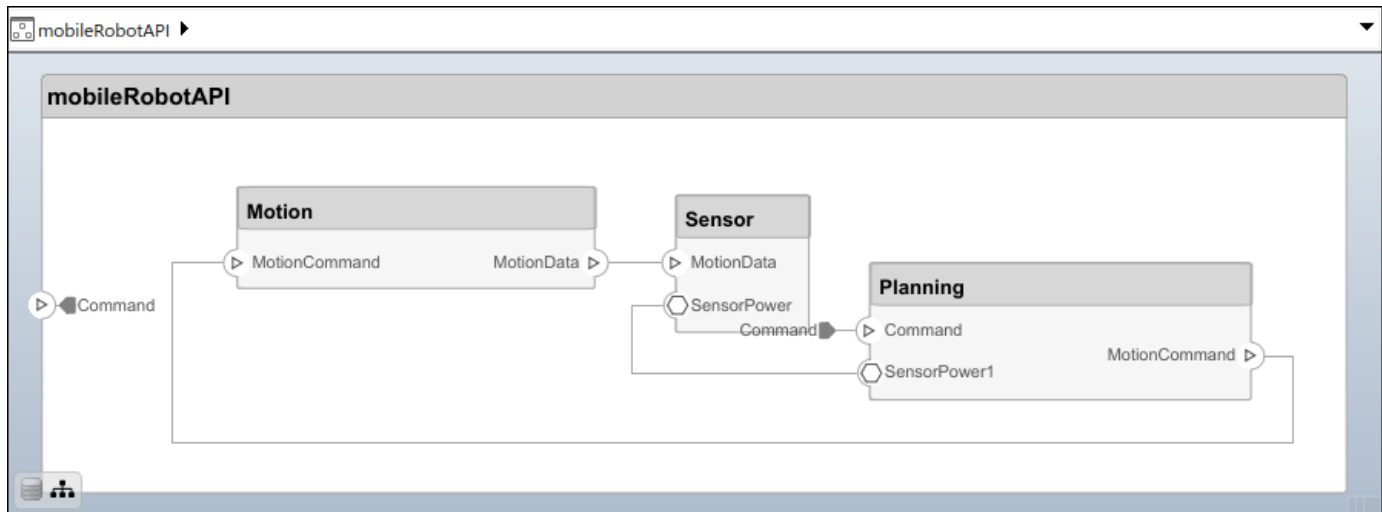
Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```





## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType,'ID',Type="uint8");
addProperty(elemSType,'Description',Type="string");
addProperty(pCompSType,'Cost',Type="double",Units="USD");
addProperty(pCompSType,'Weight',Type="double",Units="g");
addProperty(sCompSType,'develCost',Type="double",Units="USD");
addProperty(sCompSType,'develTime',Type="double",Units="hour");
```

```
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

## Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

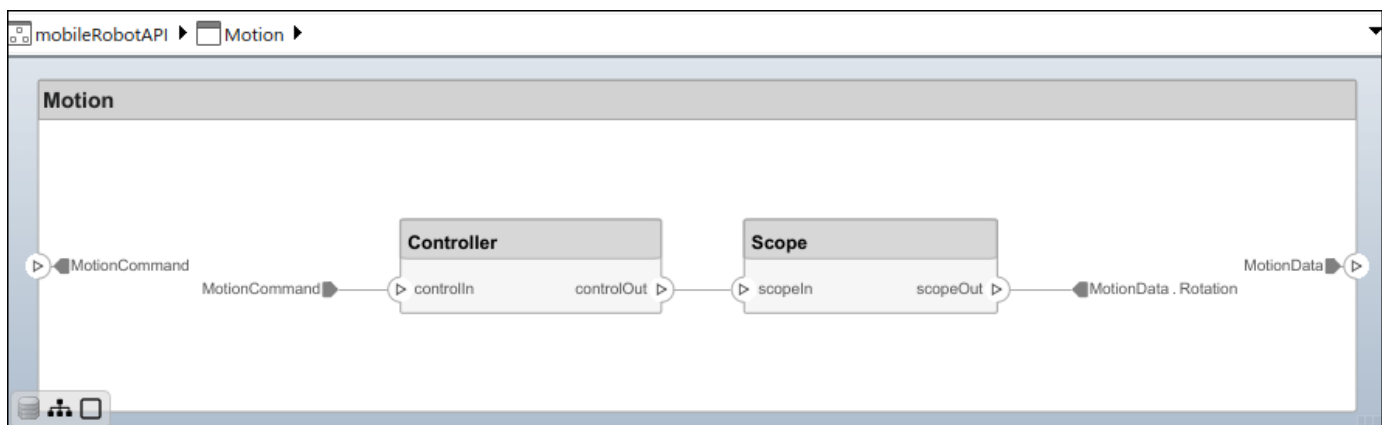
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

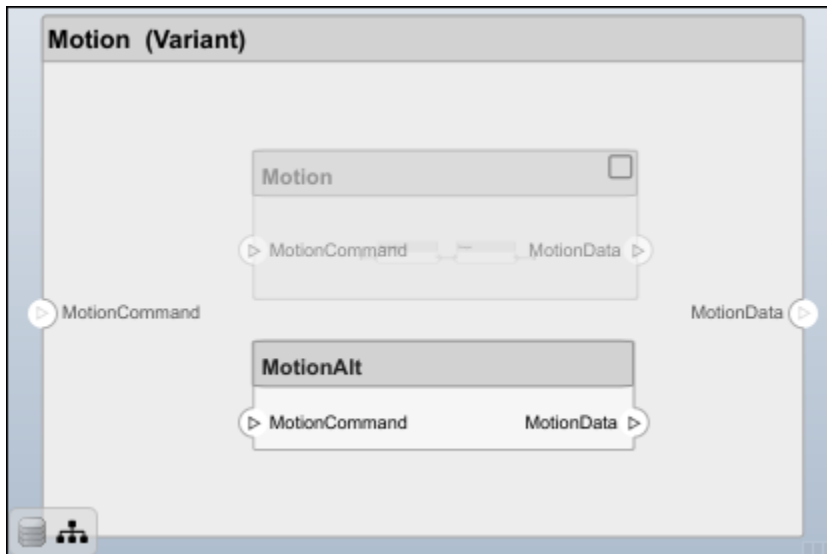
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	<p>Perform operations on a model:</p> <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> <p>A System Composer model is stored as an SLX file.</p>	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## **Version History**

**Introduced in R2019a**

### **See Also**

`createModel` | `loadModel` | `importModel` | `exportModel` | `openModel` |  
`createArchitectureModel`

### **Topics**

“Create Architecture Model”

# systemcomposer.arch.Parameter

Parameter in System Composer

## Description

A `Parameter` object describes a parameter in System Composer. Set the default properties of a parameter by setting the `Type` property. To edit and view the instance-specific parameters specified as model arguments on a component, architecture, or reference model, change the `Value` and `Unit` properties of each `Parameter` object.

## Creation

Create a `Parameter` object using the `addParameter` function on a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, or `systemcomposer.arch.Architecture` object.

## Properties

### **Name — Parameter name**

character vector | string

Parameter name, specified as a character vector or string. This property must be a valid MATLAB identifier.

Example: "advanceSpeed"

Data Types: char | string

### **Value — Parameter value**

character vector | string

Parameter value, specified as a character vector or string.

Example: "120"

Data Types: char | string

### **Unit — Parameter unit**

character vector | string

Parameter unit, specified as a character vector or string.

Example: "mph"

Data Types: char | string

### **Type — Type of parameter**

value type object

Type of parameter, specified as a `systemcomposer.ValueType` object.



**Parent — Parent architecture or component that owns parameter**

component object | variant component object | architecture object

Parent architecture or component that owns parameter, specified as a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, or `systemcomposer.arch.Architecture` object.

**Object Functions**

<code>getParameterPromotedFrom</code>	Get source parameter promoted from
<code>resetToDefault</code>	Resets parameter value to default
<code>destroy</code>	Remove model element

**Examples****Modify Parameters for Axle Architecture**

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

```
paramPressure.Type
```

```
ans =
    ValueType with properties:
```

```
    Name: 'Pressure'
```

```
    DataType: 'double'  
    Dimensions: '[1 1]'  
    Units: 'psi'  
    Complexity: 'real'  
    Minimum: ''  
    Maximum: ''  
    Description: ''  
    Owner: [1x1 systemcomposer.arch.Architecture]  
    Model: [1x1 systemcomposer.arch.Model]  
    UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'  
    ExternalUUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'31'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    0
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

Get the LeftWheel component parameter values.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))
end

paramName =
"Diameter"

paramValue =
'16'

paramUnits =
'in'

isDefault = logical
    1

paramName =
"Pressure"

paramValue =
'32'

paramUnits =
'psi'

isDefault = logical
    1

paramName =
"Wear"

paramValue =
'0.25'

paramUnits =
'in'

isDefault = logical
    1

```

First, check the evaluated RightWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))
end

paramName =
"Diameter"

paramValue = 16

paramUnits =
'in'

paramName =
"Pressure"

```

```
paramValue = 31
paramUnits =
'psi'
paramName =
"Wear"
paramValue = 0.2500
paramUnits =
'in'
```

Check the evaluated LeftWheel parameters.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
paramValue = 16
paramUnits =
'in'
paramName =
"Pressure"
paramValue = 32
paramUnits =
'psi'
paramName =
"Wear"
paramValue = 0.2500
paramUnits =
'in'
```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
paramValue =
'32'
paramUnits =
'psi'
isDefault = logical
    1
```

Update the values for the pressure on LeftWheel.

```

leftWheelComp.setParameterValue("Pressure", "34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'34'

paramUnits =
'psi'

isDefault = logical
           0

```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```

[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

isDefault = logical
           1

```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```

pressureParam.Value = "30";
pressureParam

pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'

```

Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```

sourceParam =
  Parameter with properties:

```

```
Name: 'Pressure'  
Value: '30'  
Type: [1x1 systemcomposer.ValueType]  
Parent: [1x1 systemcomposer.arch.Component]  
Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);  
pressureParam
```

```
pressureParam =  
  Parameter with properties:  
  
    Name: "LeftWheel.Pressure"  
    Value: '32'  
    Type: [1x1 systemcomposer.ValueType]  
    Parent: [1x1 systemcomposer.arch.Architecture]  
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");  
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;  
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce  
  
noiseReduce =  
  Parameter with properties:  
  
    Name: "noiseReduction"  
    Value: '30'  
    Type: [1x1 systemcomposer.ValueType]  
    Parent: [1x1 systemcomposer.arch.Architecture]  
    Unit: 'dB'
```

Rearrange the mAxleArch architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the Muffler component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");
save(model)
save(topModel)
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"



Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022b

### See Also

`addParameter` | `getParameter` | `getEvaluatedParameterValue` | `getParameterNames` | `setParameterValue` | `getParameterValue` | `setUnit` | `resetParameterToDefault`

### Topics

"Author Parameters in System Composer Using Parameter Editor"

"Access Model Arguments as Parameters on Reference Components"

"Use Parameters to Store Instance Values with Components"

# systemcomposer.arch.PhysicalConnector

Connector between physical ports

## Description

A `PhysicalConnector` object represents a connector between physical ports for a System Composer model. This class inherits from `systemcomposer.arch.BaseConnector`. This class is derived from `systemcomposer.arch.Element`.

## Creation

Create physical connectors using the `connect` function.

```
physConns = connect(architecture,physPortsA,physPortsB)
```

## Properties

### Name — Name of connector

character vector

Name of connector, specified as a character vector.

Example: `'newConnector'`

Data Types: `char`

### Parent — Architecture that owns connector

architecture object

Architecture that owns connector, specified as a `systemcomposer.arch.Architecture` object.

### Ports — Ports of connection

array of port objects

Ports of connection, specified as an array of `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` objects.

### UUID — Universal unique identifier

character vector

Universal unique identifier for model connector, specified as a character vector.

Example: `'91d5de2c-b14c-4c76-a5d6-5dd0037c52df'`

Data Types: `char`

### ExternalUUID — Unique external identifier

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the model connector and through all operations that preserve the UUID.

Data Types: char

**Model – Parent model**

model object

Parent System Composer model of connector, specified as a `systemcomposer.arch.Model` object.

**SimulinkHandle – Simulink handle**

numeric value

Simulink handle, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: `double`

**SimulinkModelHandle – Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a `double`.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: `double`

**Object Functions**

<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>getSourceElement</code>	Gets data elements selected on source port for connection
<code>getDestinationElement</code>	Gets data elements selected on destination port for connection
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>getQualifiedName</code>	Get model element qualified name
<code>destroy</code>	Remove model element

**Examples****Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

## Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

## Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

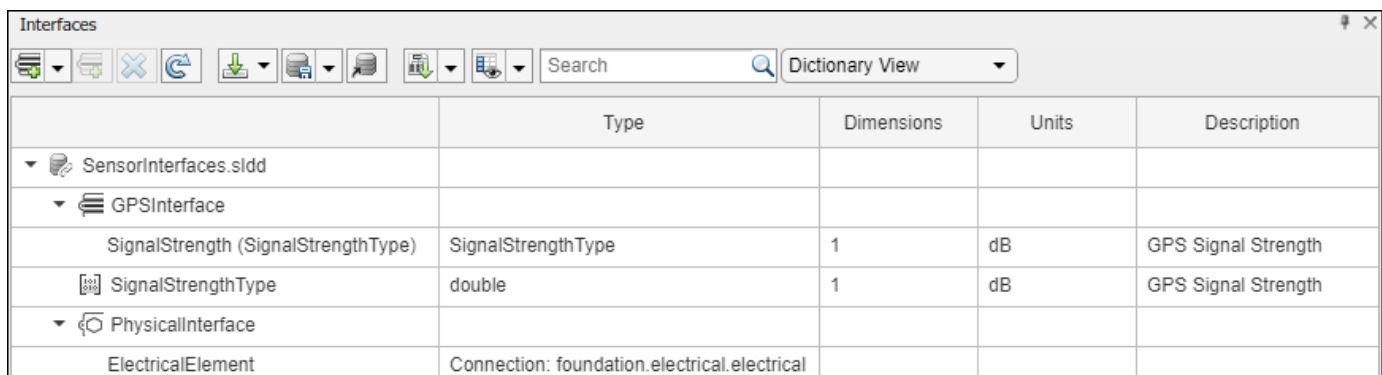
Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.



The screenshot shows the 'Interfaces' editor window with a toolbar and a table view. The table has columns for Type, Dimensions, Units, and Description. The tree view shows the following structure:

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sldd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch, "Sensor");
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, ...
    {'in', 'physical'});
sensorPorts(2).setInterface(physicalInterface)

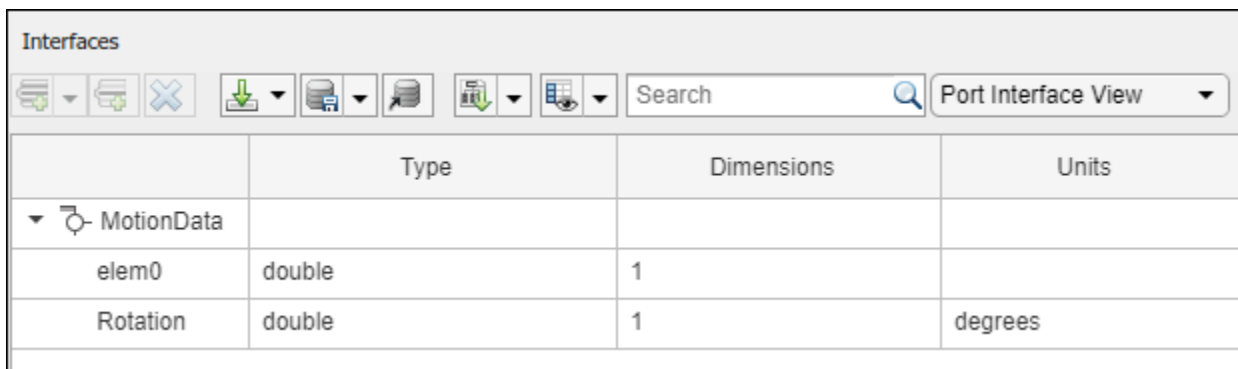
componentPlanning = addComponent(arch, "Planning");
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorPower1', 'MotionCommand'}, ...
    {'in', 'physical', 'out'});
planningPorts(2).setInterface(physicalInterface)



componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, ...
    {'in', 'out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces			
 <input type="text" value="Search"/> <span>Port Interface View</span>			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch, componentSensor, componentPlanning, Rule="interface");
c_motionData = connect(arch, componentMotion, componentSensor);
c_motionCommand = connect(arch, componentPlanning, componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

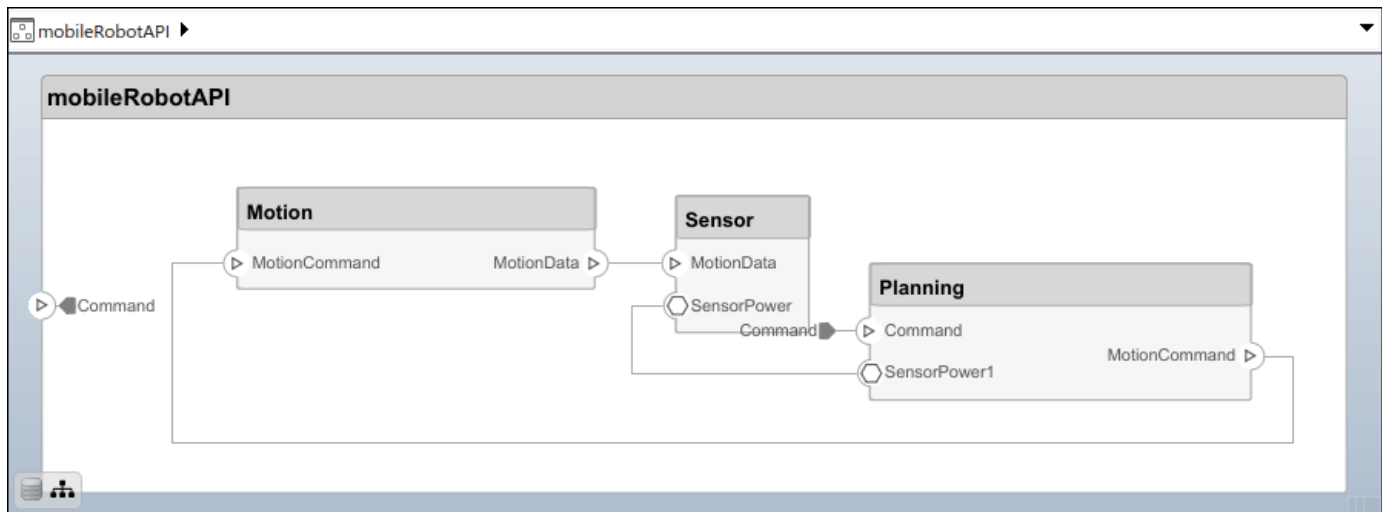
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile, "projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile, "physicalComponent", AppliesTo="Component");
sCompSType = addStereotype(profile, "softwareComponent", AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile, "standardConn", AppliesTo="Connector");
```

## Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

## Save Profile

```
profile.save;
```

## Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```



Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

### Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1), controllerCompPortIn);
```

For output connections, the data element must be specified.

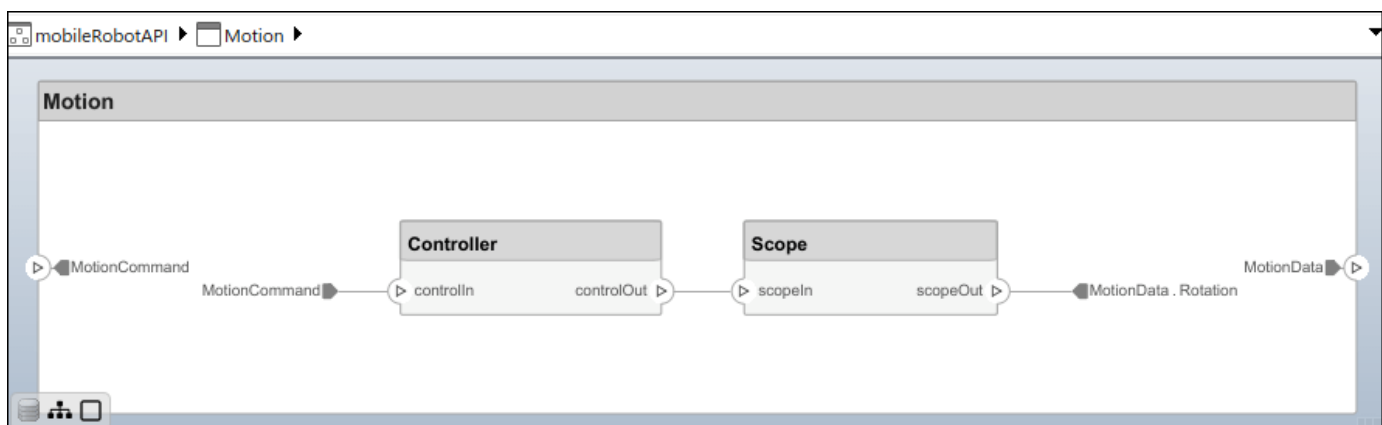
```
c_planningScope = connect(scopeCompPortOut, motionPorts(2), DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, ...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");  
referenceArch = referenceModel.Architecture;  
newComponents = addComponent(referenceArch,"Gyroscope");  
referenceModel.save
```

```
linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save  
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

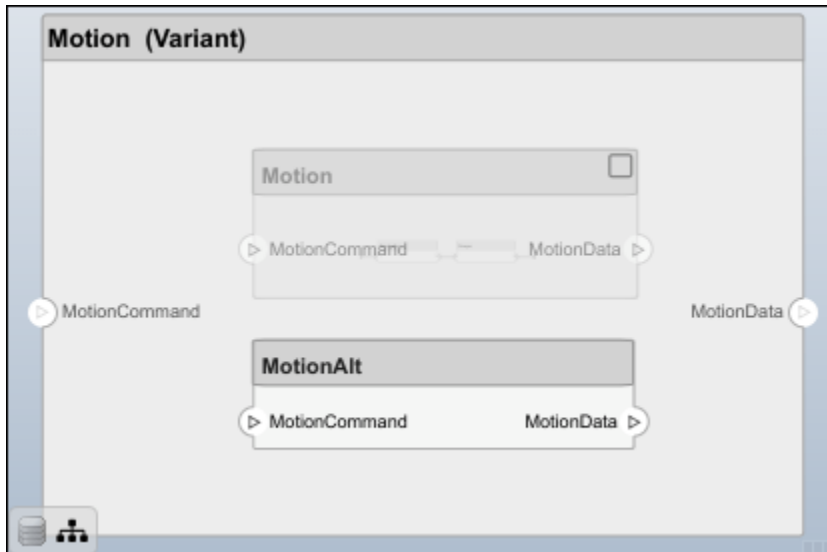
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp, "MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2021b

### See Also

`systemcomposer.arch.Element` | `systemcomposer.arch.BaseConnector` | `systemcomposer.arch.Connector` | `connect` | `Component`

**Topics**

“Create Architecture Model”

“Implement Component Behavior Using Simscape”

# systemcomposer.arch.VariantComponent

Variant component in System Composer model

## Description

A `VariantComponent` object represents a variant component that allows you to create multiple design alternatives for a component in a System Composer model. This class inherits from `systemcomposer.arch.BaseComponent`. This class is derived from `systemcomposer.arch.Element`.

## Creation

Create a variant component using the `addVariantComponent` function.

```
varComp = addVariantComponent(archObj, 'compName');
```

## Properties

### Name — Name of variant component

character vector

Name of variant component, specified as a character vector.

Data Types: char

### Position — Position of component on canvas

vector of coordinates in pixels

Position of component on canvas, specified as a vector of coordinates in pixels: [left top right bottom].

### Parent — Architecture that owns variant component

architecture object

Architecture that owns variant component, specified as a `systemcomposer.arch.Architecture` object.

### Architecture — Architecture of active variant choice

architecture object

Architecture of the active variant choice, specified as a `systemcomposer.arch.Architecture` object.

### Ports — Input and output ports

component port objects

Input and output ports of variant component, specified as `systemcomposer.arch.ComponentPort` objects.

**Parameters — Parameters of component**

array of parameter objects

Parameters of component, specified as an array of `systemcomposer.arch.Parameter` objects.

**OwnedArchitecture — Architecture owned by variant component**

architecture object

Architecture owned by variant component, specified as a `systemcomposer.arch.Architecture` object.

**OwnedPorts — Array of component ports**

array of component port objects

Array of component ports, specified as an array of `systemcomposer.arch.ComponentPort` objects.

**UUID — Universal unique identifier**

character vector

Universal unique identifier for variant component, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the variant component and through all operations that preserve the UUID.

Data Types: char

**Model — Parent model**

model object

Parent System Composer model of component, specified as a `systemcomposer.arch.Model` object.

**SimulinkHandle — Simulink handle**

numeric value

Simulink handle, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.

Example: `handle = get(object, 'SimulinkHandle')`

Data Types: double

**SimulinkModelHandle — Simulink handle to parent model**

numeric value

Simulink handle to parent System Composer model, specified as a double.

This property is necessary for several Simulink related workflows and for using Requirements Toolbox programmatic interfaces.



Example: `handle = get(object, 'SimulinkModelHandle')`

Data Types: `double`

## Object Functions

<code>addChoice</code>	Add variant choices to variant component
<code>setCondition</code>	Set condition on variant choice
<code>setActiveChoice</code>	Set active choice on variant component
<code>getChoices</code>	Get available choices in variant component
<code>getActiveChoice</code>	Get active choice on variant component
<code>getCondition</code>	Return variant control on choice within variant component
<code>isProtected</code>	Find if component reference model is protected
<code>isReference</code>	Find if component is referenced to another model
<code>connect</code>	Create architecture model connections
<code>getPort</code>	Get port from component
<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>removeStereotype</code>	Remove stereotype from model element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>getQualifiedName</code>	Get model element qualified name
<code>getParameter</code>	Get parameter from architecture or component
<code>getEvaluatedParameterValue</code>	Get evaluated value of parameter from element
<code>getParameterNames</code>	Get parameter names on element
<code>getParameterValue</code>	Get value of parameter
<code>setParameterValue</code>	Set value of parameter
<code>setUnit</code>	Set units on parameter value
<code>resetParameterToDefault</code>	Reset parameter on component to default value
<code>destroy</code>	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

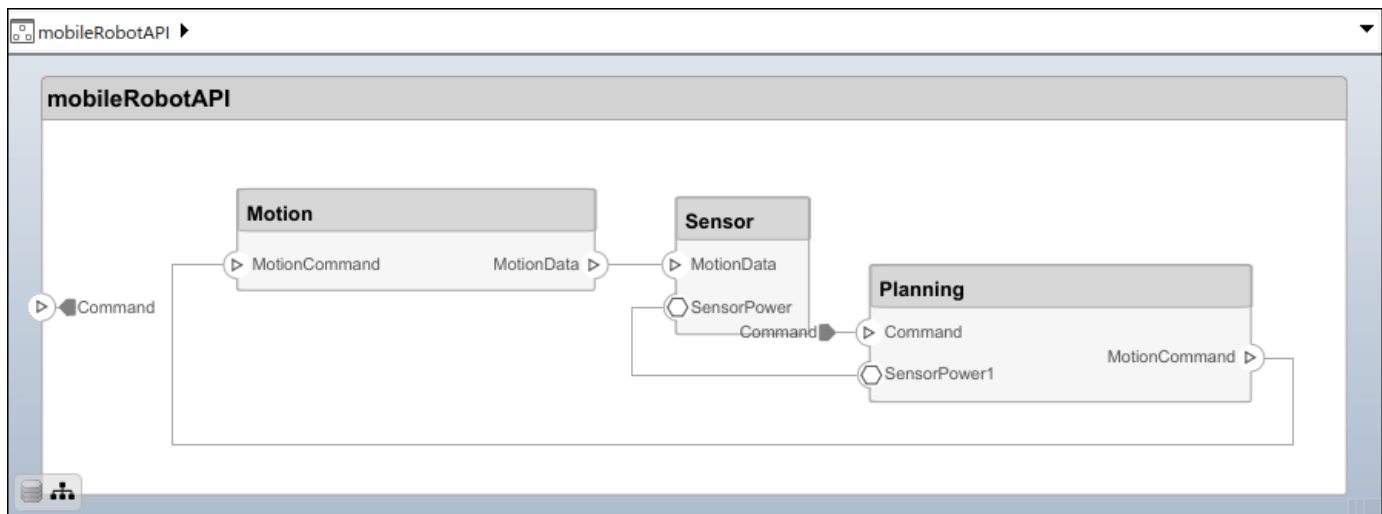
	Type	Dimensions	Units	Description
SensorInterfaces.slidd				
GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
```





### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

#### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

#### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType,'ID',Type="uint8");
addProperty(elemSType,'Description',Type="string");
addProperty(pCompSType,'Cost',Type="double",Units="USD");
addProperty(pCompSType,'Weight',Type="double",Units="g");
addProperty(sCompSType,'develCost',Type="double",Units="USD");
addProperty(sCompSType,'develTime',Type="double",Units="hour");
```

```

addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");

```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```

applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")

```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```

batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");

```

Set properties for each component.

```

setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
  'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
  'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
  'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

## Add Hierarchy

Add two components named `Controller` and `Scope` inside the `Motion` component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

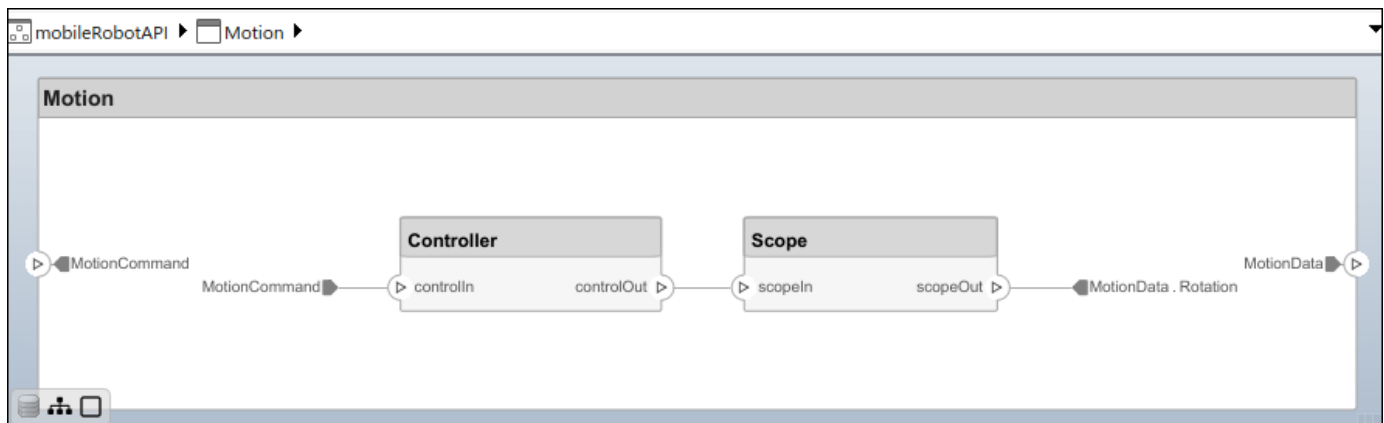
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

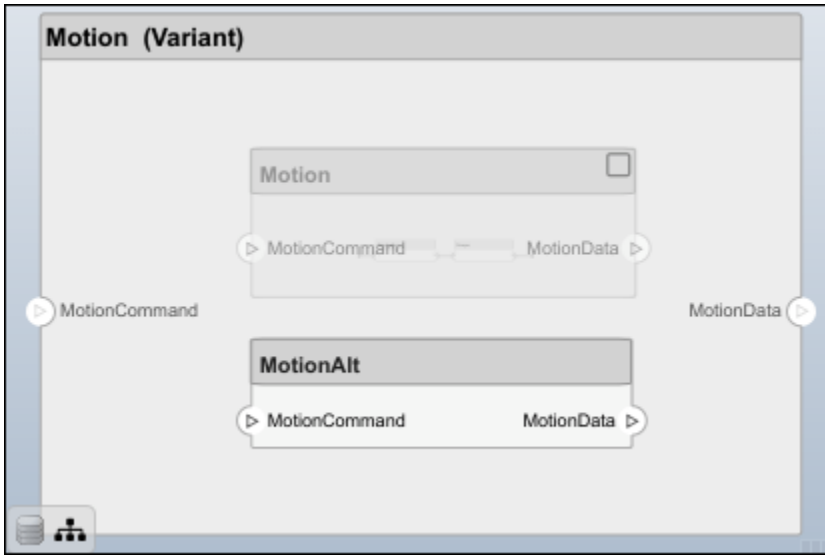
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>



Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## **Version History**

**Introduced in R2019a**

### **See Also**

Variant Component

### **Topics**

"Decompose and Reuse Components"

# systemcomposer.interface.DataElement

Data element in data interface

## Description

A DataElement object represents a data element in a data interface.

## Creation

Create a data element using the addElement function.

```
element = addElement(interface, 'newElement')
```

## Properties

### Interface — Parent data interface of data element

data interface object

Parent data interface of data element, specified as a systemcomposer.interface.DataInterface object.

### Name — Data element name

character vector | string

Data element name, specified as a character vector or string.

Example: 'newElement'

Data Types: char | string

### Type — Type of data element

data interface object | value type object

Type of data element, specified as a systemcomposer.interface.DataInterface or systemcomposer.ValueType object.

### Dimensions — Dimensions of data element

character vector | string

Dimensions of data element, specified as a character vector or string.

Data Types: char | string

### Description — Description of data element

character vector | string

Description of data element, specified as a character vector or string.

Data Types: char | string

**UUID – Universal unique identifier**

character vector

Universal unique identifier for data element, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUUID – Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the data element and through all operations that preserve the UUID.

Data Types: char

**Object Functions**

setName	Set name for value type, function argument, interface, or element
setType	Set shared type on data element or function argument
setDimensions	Set dimensions for value type
setUnits	Set units for value type
setComplexity	Set complexity for value type
setMinimum	Set minimum for value type
setMaximum	Set maximum for value type
setDescription	Set description for value type or interface
createOwnedType	Create owned value type on data element or function argument
destroy	Remove model element

**Examples****Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

**Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

**Add Components, Ports, Connections, and Interfaces**

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");  
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```

dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.sldd");

```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sldd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```

componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)

```

```

componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)

```

```

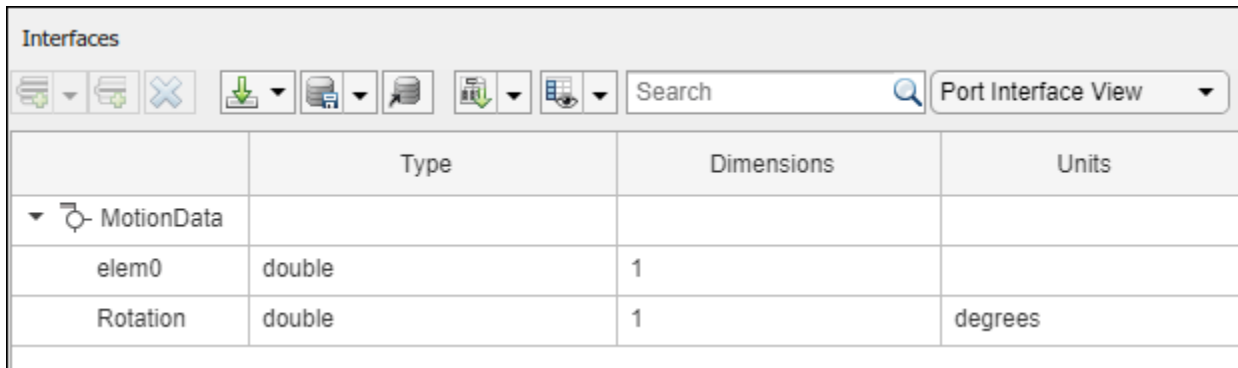
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});



```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces				
				
		Type	Dimensions	Units
▼	 MotionData			
	elem0	double	1	
	Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch,"Command","in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

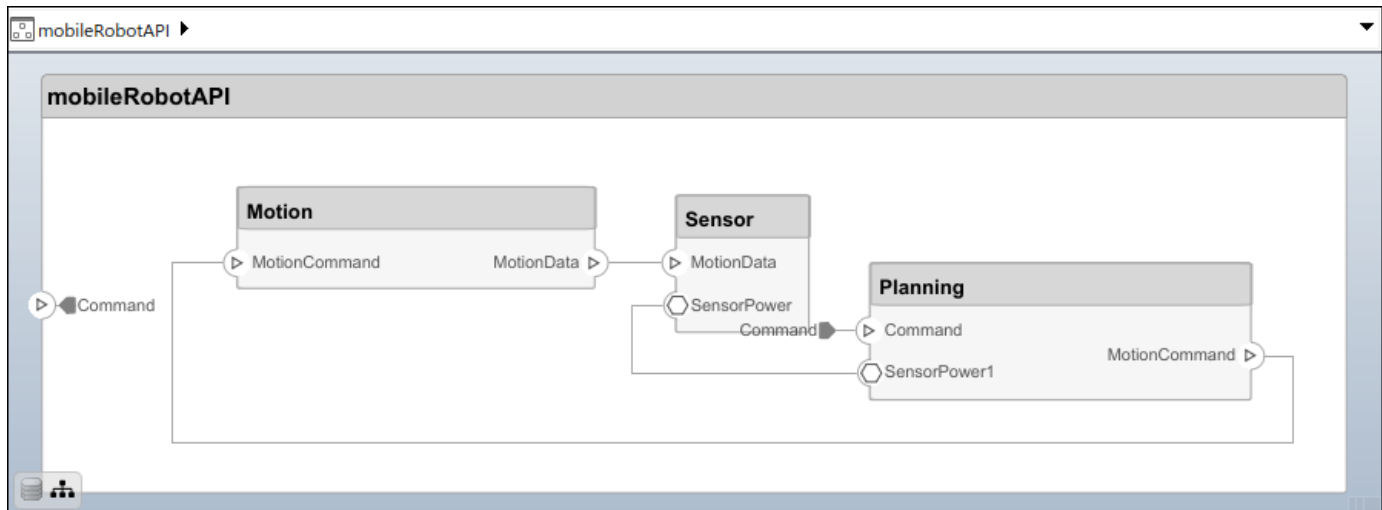
```
compPort = getPort(componentPlanning,"Command");
c_Command = connect(archPort,compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType,'ID',Type="uint8");
addProperty(elemSType,'Description',Type="string");
addProperty(pCompSType,'Cost',Type="double",Units="USD");
addProperty(pCompSType,'Weight',Type="double",Units="g");
addProperty(sCompSType,'develCost',Type="double",Units="USD");
addProperty(sCompSType,'develTime',Type="double",Units="hour");
```

```
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```



## Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

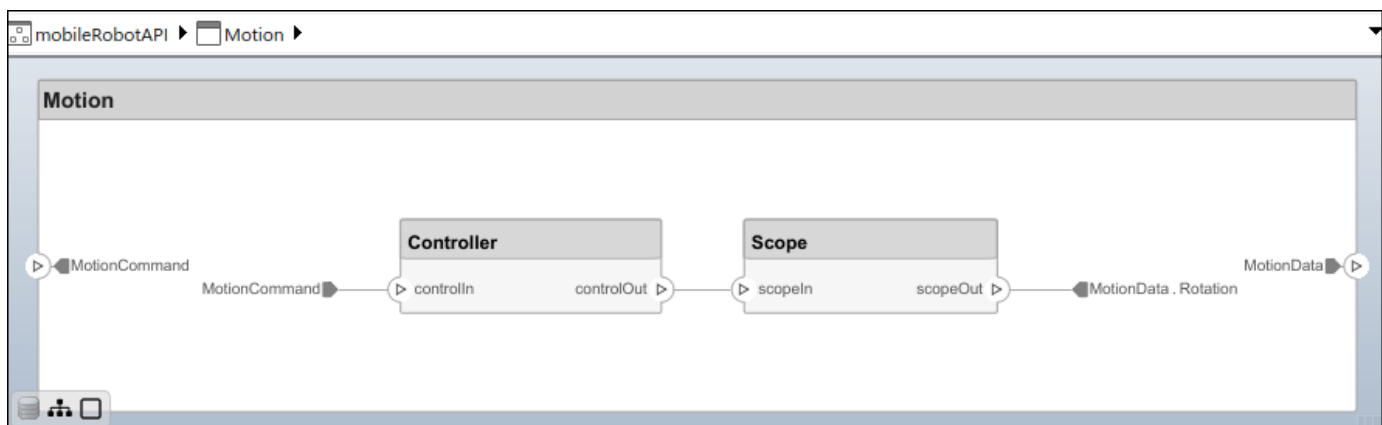
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

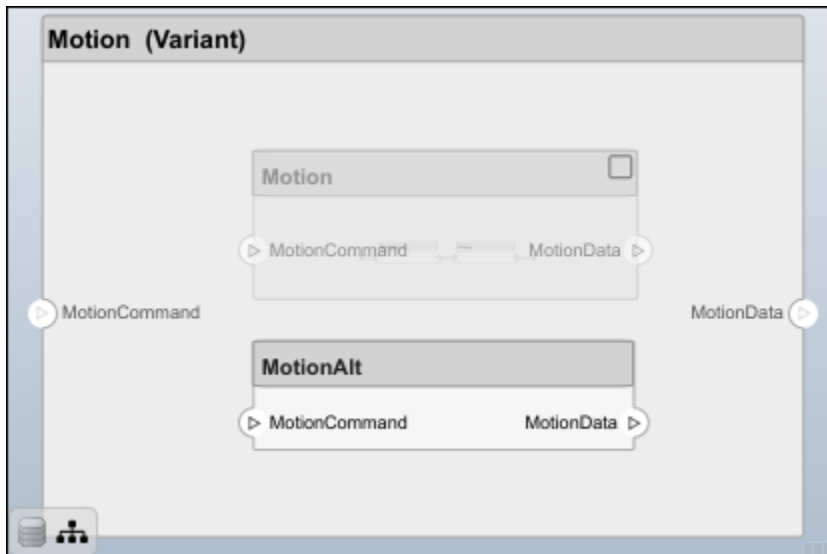
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanupArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>“Manage Interfaces with Data Dictionaries”</li> <li>“Reference Data Dictionaries”</li> </ul>

Term	Definition	Application	More Information
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`addElement` | `removeElement` | `getElement` | `systemcomposer.ValueType` | `systemcomposer.interface.Dictionary` | `systemcomposer.interface.DataInterface`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

# systemcomposer.interface.DataInterface

Data interface

## Description

A `DataInterface` object represents the structure of a data interface.

## Creation

Create a data interface using the `addInterface` function.

```
interface = addInterface(dictionary, 'newInterface')
```

## Properties

### Owner — Parent of data interface

dictionary object | data element object | architecture port object

Parent of data interface, specified as a `systemcomposer.interface.Dictionary`, `systemcomposer.interface.DataElement`, or `systemcomposer.arch.ArchitecturePort` object.

### Model — Parent model

model object

Parent System Composer model of data interface, specified as a `systemcomposer.arch.Model` object.

### Name — Data interface name

character vector | string

Data interface name, specified as a character vector or string. This property must be a valid MATLAB identifier.

Example: `'newInterface'`

Data Types: `char` | `string`

### Elements — Elements in interface

array of data element objects

Elements in interface, specified as an array of `systemcomposer.interface.DataElement` objects.

### Description — Data interface description

character vector | string

Data interface description, specified as a character vector or string.

Data Types: `char` | `string`

**UUID – Universal unique identifier**

character vector

Universal unique identifier for data interface, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUID – Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the data interface and through all operations that preserve the UUID.

Data Types: char

**Object Functions**

addElement	Add element
getElement	Get object for element
removeElement	Remove element
setName	Set name for value type, function argument, interface, or element
setDescription	Set description for value type or interface
applyStereotype	Apply stereotype to architecture model element
getStereotypes	Get stereotypes applied on element of architecture model
getStereotypeProperties	Get stereotype property names on element
removeStereotype	Remove stereotype from model element
getProperty	Get property value corresponding to stereotype applied to element
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from element
setProperty	Set property value corresponding to stereotype applied to element
hasStereotype	Find if element has stereotype applied
hasProperty	Find if element has property
destroy	Remove model element

**Examples****Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

**Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

**Add Components, Ports, Connections, and Interfaces**

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
SensorInterfaces.slidd				
GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
```



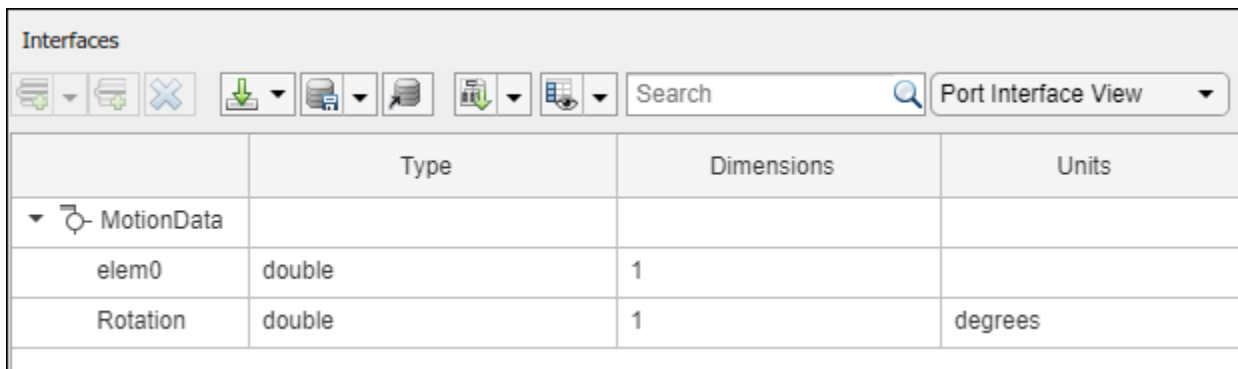
```
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, ...
    {'in', 'out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch, componentSensor, componentPlanning, Rule="interface");
c_motionData = connect(arch, componentMotion, componentSensor);
c_motionCommand = connect(arch, componentPlanning, componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

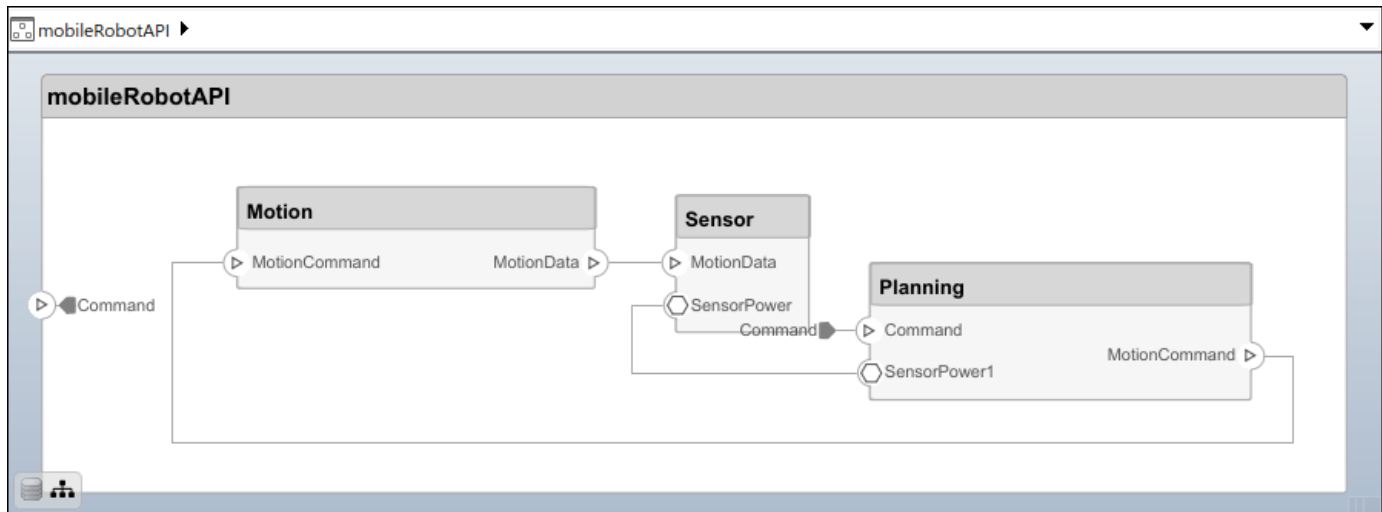
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

#### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

#### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
```

```

addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");

```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```

applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")

```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```

batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");

```

Set properties for each component.

```

setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

## Add Hierarchy

Add two components named `Controller` and `Scope` inside the `Motion` component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

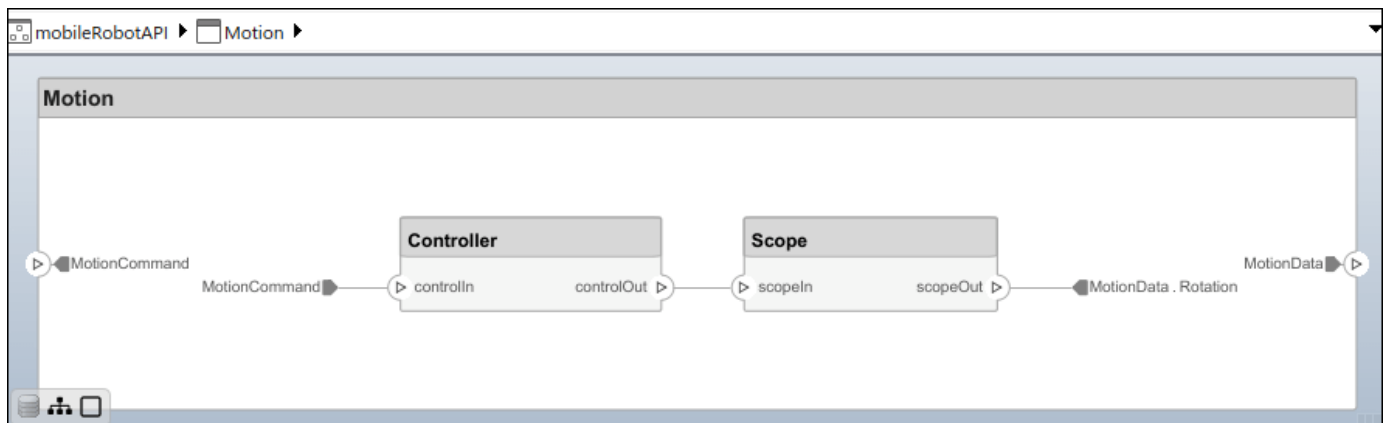
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

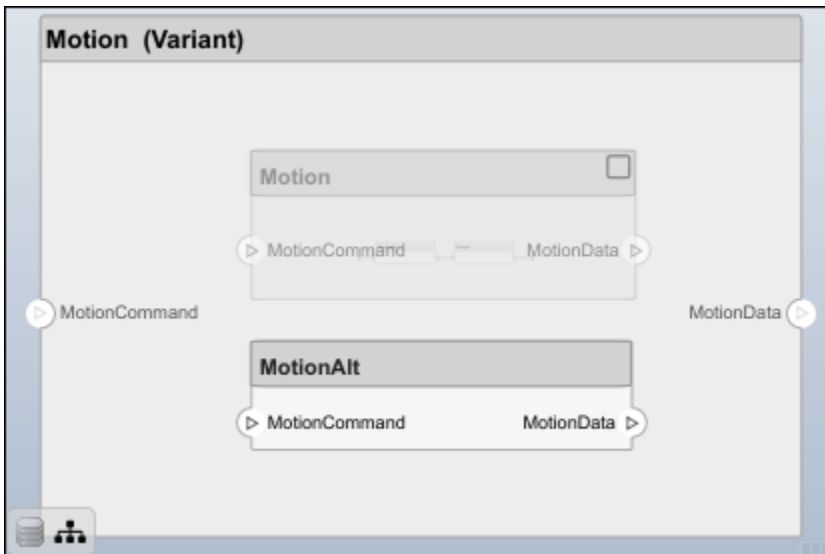
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>

Term	Definition	Application	More Information
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`createInterface` | `setInterface` | `addInterface` | `getInterface` | `getInterfaceNames` | `removeInterface` | `systemcomposer.ValueType` | `systemcomposer.interface.Dictionary` | `systemcomposer.interface.DataElement`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”



# systemcomposer.interface.Dictionary

Interface data dictionary of architecture model

## Description

A Dictionary object represents the interface data dictionary of a System Composer model.

## Creation

Create an interface data dictionary using the `systemcomposer.createDictionary` function.

```
dictionary = systemcomposer.createDictionary('newDictionary.sldd');
```

## Properties

### Interfaces — Interfaces defined in dictionary

array of interface objects

Interfaces defined in dictionary, specified as an array of `systemcomposer.interface.DataInterface`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` objects.

### Profiles — Profiles attached to dictionary

array of profile objects

Profiles attached to dictionary, specified as an array of `systemcomposer.profile.Profile` objects.

### UUID — Universal unique identifier

character vector

Universal unique identifier for interface data dictionary, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### ExternalUUID — Unique external identifier

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the interface data dictionary and through all operations that preserve the UUID.

Data Types: char

## Object Functions

`addValueType` Create named value type in interface dictionary

<code>addInterface</code>	Create named data interface in interface dictionary
<code>addPhysicalInterface</code>	Create named physical interface in interface dictionary
<code>addServiceInterface</code>	Create named service interface in interface dictionary
<code>getInterface</code>	Get object for named interface in interface dictionary
<code>getInterfaceNames</code>	Get names of all interfaces in interface dictionary
<code>removeInterface</code>	Remove named interface from interface dictionary
<code>applyProfile</code>	Apply profile to model
<code>removeProfile</code>	Remove profile from model
<code>save</code>	Save architecture model or data dictionary
<code>saveToDictionary</code>	Save interfaces to dictionary
<code>addReference</code>	Add reference to dictionary
<code>removeReference</code>	Remove reference to dictionary
<code>destroy</code>	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");  
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");  
interface = dictionary.addInterface("GPSInterface");  
element = interface.addElement("SignalStrength");  
valueType = dictionary.addValueType("SignalStrengthType", Units="dB", ...  
    Description="GPS Signal Strength");  
element.setType(valueType);  
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");  
physicalElement = addElement(physicalInterface, "ElectricalElement", ...  
    Type="electrical.electrical");  
linkDictionary(model, "SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch, "Sensor");
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, ...
    {'in', 'physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch, "Planning");
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorPower1', 'MotionCommand'}, ...
    {'in', 'physical', 'out'});
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, ...
    {'in', 'out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.

Interfaces			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

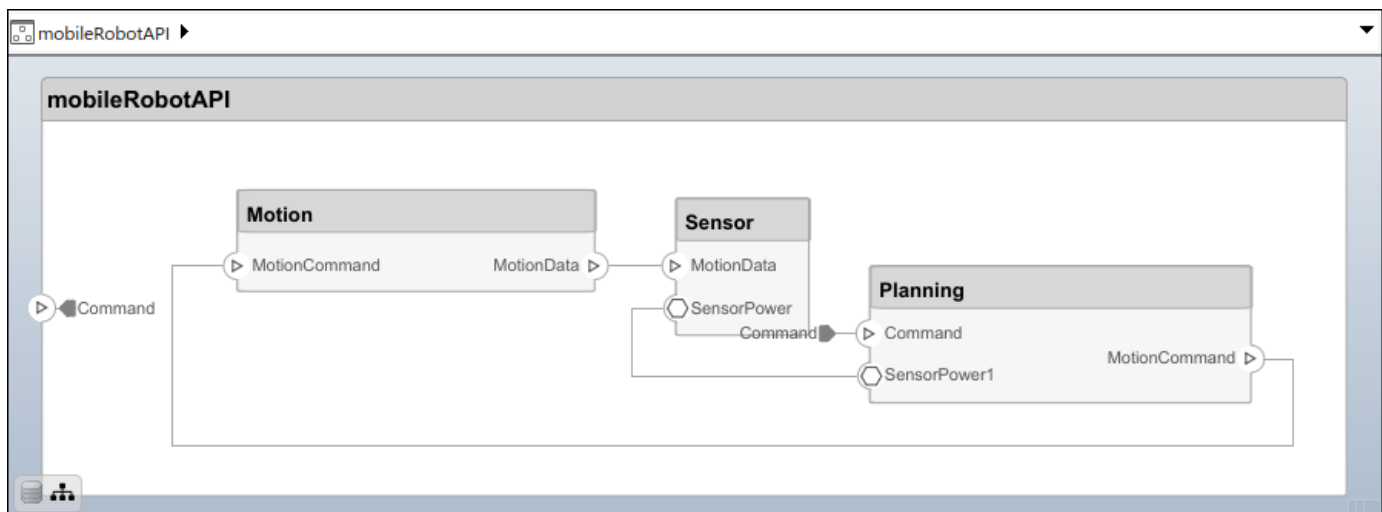
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

### Add Hierarchy

Add two components named Controller and Scope inside the Motion component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);
```

For outport connections, the data element must be specified.

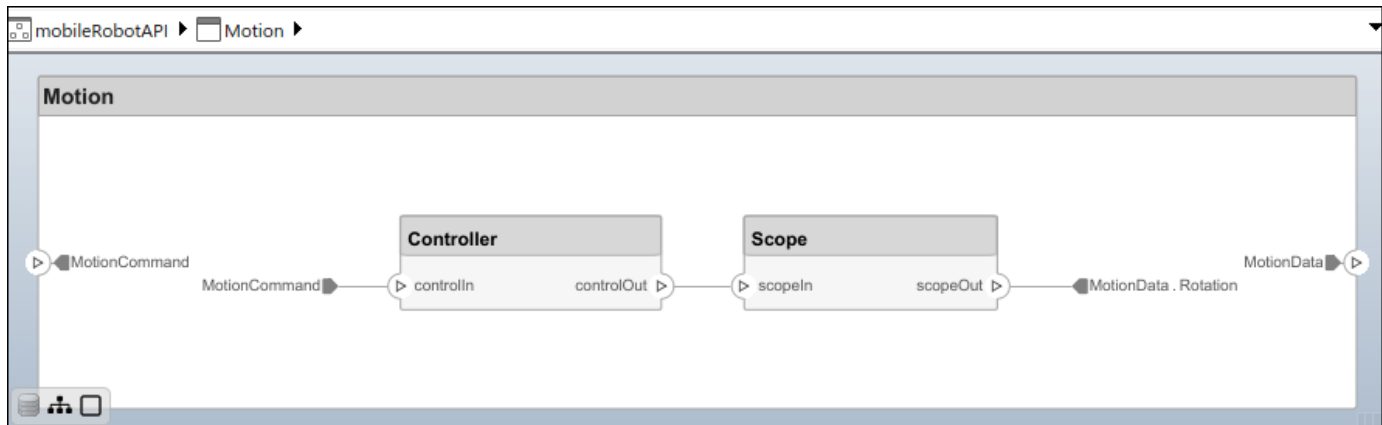
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save
```

```
linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the `Planning` component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

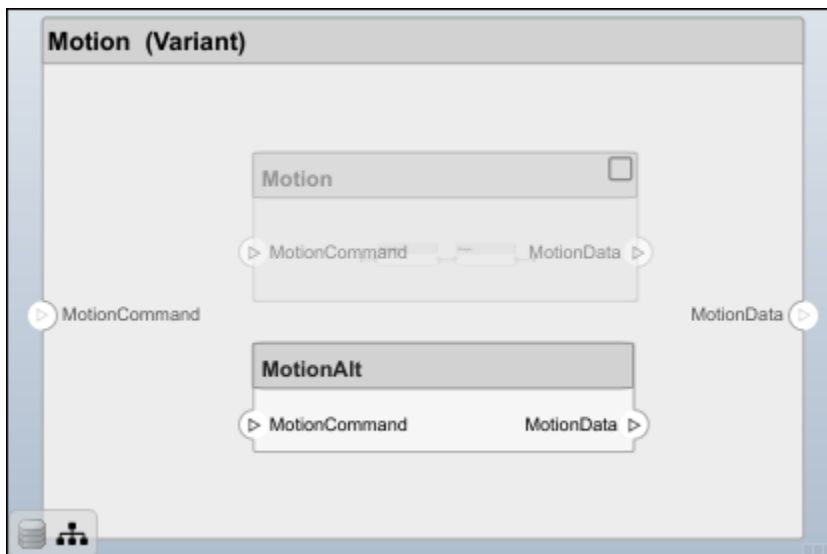
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```



## Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	"Create Value Types as Interfaces"
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	"Define Owned Interfaces Local to Ports"
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the "Interface Adapter" dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• "Interface Adapter"</li> <li>• Adapter</li> </ul>

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>• “Author Software Architectures”</li> <li>• “Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

## Version History

Introduced in R2019a

### See Also

`openDictionary` | `createDictionary` | `saveToDictionary` | `systemcomposer.ValueType` | `systemcomposer.interface.DataElement` | `systemcomposer.interface.DataInterface` | `systemcomposer.interface.PhysicalInterface` | `systemcomposer.interface.PhysicalElement` | `systemcomposer.interface.PhysicalDomain` | `systemcomposer.interface.ServiceInterface` | `systemcomposer.interface.FunctionElement`

### Topics

[“Create Interfaces”](#)  
[“Manage Interfaces with Data Dictionaries”](#)  
[“Specify Physical Interfaces on Ports”](#)  
[“Client-Server Interfaces in Class Diagram View”](#)

# systemcomposer.interface.FunctionArgument

Function argument in function element in client-server interface

## Description

A FunctionArgument object describes the attributes of an argument in a function element `systemcomposer.interface.FunctionElement`.

## Creation

Set a function prototype using the `setFunctionPrototype` function and then get a function argument using the `getFunctionArgument` function.

```
setFunctionPrototype(element, "y=f0(u)")
argument = getFunctionArgument(functionElement, "y")
```

## Properties

### Interface — Parent service interface of function argument

service interface object

Parent service interface of function argument, specified as a `systemcomposer.interface.ServiceInterface` object.

### Name — Function argument name

character vector | string

Function argument name, specified as a character vector or string.

Example: "y"

Data Types: char | string

### Type — Type of function argument

value type object

Type of function argument, specified as a `systemcomposer.ValueType` object.

### Dimensions — Dimensions of function argument

character vector | string

Dimensions of function argument, specified as a character vector or string.

Data Types: char | string

### Description — Description of function argument

character vector | string

Description of function argument, specified as a character vector or string.

Data Types: char | string

**UUID – Universal unique identifier**

character vector

Universal unique identifier for function argument, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUUID – Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the function argument and through all operations that preserve the UUID.

Data Types: char

**Object Functions**

setName	Set name for value type, function argument, interface, or element
setType	Set shared type on data element or function argument
setDimensions	Set dimensions for value type
setUnits	Set units for value type
setComplexity	Set complexity for value type
setMinimum	Set minimum for value type
setMaximum	Set maximum for value type
setDescription	Set description for value type or interface
createOwnedType	Create owned value type on data element or function argument
destroy	Remove model element

**Examples****Get Function Argument**

Create a new model.

```
model = systemcomposer.createModel("archModel","SoftwareArchitecture",true);
```

Create a service interface.

```
interface = addServiceInterface(model.InterfaceDictionary,"newServiceInterface");
```

Create a function element.

```
element = addElement(interface,"newFunctionElement");
```

Set a function prototype to add function arguments.

```
setFunctionPrototype(element,"y=f0(u)");
```

Get a function argument.

```
argument = getFunctionArgument(element,"y")
```

```
argument =
```

```
FunctionArgument with properties:
```



```

Interface: [1x1 systemcomposer.interface.ServiceInterface]
Element: [1x1 systemcomposer.interface.FunctionElement]
  Name: 'y'
  Type: [1x1 systemcomposer.ValueType]
Dimensions: '1'
Description: ''
  UUID: '018b4e55-fa8f-4250-ac2b-df72bf620feb'
ExternalUID: ''

```

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”

Term	Definition	Application	More Information
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution — When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> <li>• Asynchronous execution — When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul>	<code>systemcomposer.interface.FunctionElement</code>
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	<code>systemcomposer.interface.FunctionArgument</code>

Term	Definition	Application	More Information
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2022a

## **See Also**

`addElement` | `removeElement` | `getElement` | `systemcomposer.interface.Dictionary`

## **Topics**

*“Author Service Interfaces for Client-Server Communication”*

*“Client-Server Interfaces in Class Diagram View”*

*“Define Port Interfaces Between Components”*

# systemcomposer.interface.FunctionElement

Function in client-server interface

## Description

A `FunctionElement` object describes the attributes of a function in a client-server interface `systemcomposer.interface.ServiceInterface`.

## Creation

Create a function element using the `addElement` function.

```
element = addElement(serviceInterface, "f0")
```

## Properties

### Interface — Parent service interface of function element

service interface object

Parent service interface of function element, specified as a `systemcomposer.interface.ServiceInterface` object.

### Name — Function element name

character vector | string

Function element name, specified as a character vector or string.

Example: `"newFunctionElement"`

Data Types: `char` | `string`

### Asynchronous — Whether function element is asynchronous

`true` or `1` | `false` or `0`

Whether function element is asynchronous, specified as a logical.

Data Types: `logical`

### FunctionPrototype — Function prototype

character vector | string

Function prototype to define input and output arguments, specified as a character vector or string.

Example: `"[y1,y2]=f1(u1,u2)"`

Data Types: `char` | `string`

### FunctionArguments — Function arguments

array of function argument objects

Function arguments, specified as an array of `systemcomposer.interface.FunctionArgument` objects.

### UUID – Universal unique identifier

character vector

Universal unique identifier for function element, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### ExternalUUID – Unique external identifier

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the function element and through all operations that preserve the **UUID**.

Data Types: char

## Object Functions

<code>setName</code>	Set name for value type, function argument, interface, or element
<code>setFunctionPrototype</code>	Set prototype for function element
<code>getFunctionArgument</code>	Get function argument on function element
<code>setAsynchronous</code>	Set function element as asynchronous
<code>destroy</code>	Remove model element

## Examples

### Get Function Argument

Create a new model.

```
model = systemcomposer.createModel("archModel", "SoftwareArchitecture", true);
```

Create a service interface.

```
interface = addServiceInterface(model.InterfaceDictionary, "newServiceInterface");
```

Create a function element.

```
element = addElement(interface, "newFunctionElement");
```

Set a function prototype to add function arguments.

```
setFunctionPrototype(element, "y=f0(u)")
```

Get a function argument.

```
argument = getFunctionArgument(element, "y")
```

```
argument =
```

FunctionArgument with properties:

```
Interface: [1x1 systemcomposer.interface.ServiceInterface]
```

```

Element: [1x1 systemcomposer.interface.FunctionElement]
  Name: 'y'
  Type: [1x1 systemcomposer.ValueType]
Dimensions: '1'
Description: ''
  UUID: '018b4e55-fa8f-4250-ac2b-df72bf620feb'
ExternalUID: ''

```

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”



Term	Definition	Application	More Information
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution — When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> <li>• Asynchronous execution — When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul>	<code>systemcomposer.interface.FunctionElement</code>
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: <b>Type</b> , <b>Dimensions</b> , <b>Units</b> , <b>Complexity</b> , <b>Minimum</b> , <b>Maximum</b> , and <b>Description</b> .	<code>systemcomposer.interface.FunctionArgument</code>

Term	Definition	Application	More Information
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the “Interface Adapter” dialog box: <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2022a

**See Also**

`addElement` | `removeElement` | `getElement` | `systemcomposer.interface.Dictionary`

**Topics**

“Author Service Interfaces for Client-Server Communication”

“Client-Server Interfaces in Class Diagram View”

“Define Port Interfaces Between Components”

# systemcomposer.interface.PhysicalDomain

Physical domain in System Composer

## Description

A `PhysicalDomain` object describes a physical domain in System Composer. A physical domain can be used as an owned interface on a port and typed to a physical element on a physical interface.

## Creation

Create an owned interface using a physical domain on a port.

```
model = systemcomposer.createModel('archModel',true);
rootArch = get(model,'Architecture');
newComponent = addComponent(rootArch,'newComponent');
newPort = addPort(newComponent.Architecture,'newCompPort','physical');
port = newComponent.getPort('newCompPort');
interface = port.createInterface;
interface.Domain = 'mechanical.rotational.rotational'
```

## Properties

### Owner — Parent of physical domain

architecture port object

Parent of physical domain, specified as a `systemcomposer.arch.ArchitecturePort` object.

### Model — Parent model

model object

Parent System Composer model of physical domain, specified as a `systemcomposer.arch.Model` object.

### Domain — Physical domain

character vector | string

Physical domain, specified as a character vector or string of a partial physical domain name. For a list of valid physical domain names, see “Domain-Specific Line Styles” (Simscape).

Data Types: char | string

### UUID — Universal unique identifier

character vector

Universal unique identifier for physical domain, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the physical domain and through all operations that preserve the UUID.

Data Types: char

**Object Functions**

destroy Remove model element

**Examples****Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

**Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

**Add Components, Ports, Connections, and Interfaces**

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");  
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");  
interface = dictionary.addInterface("GPSInterface");  
element = interface.addElement("SignalStrength");  
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...  
    Description="GPS Signal Strength");  
element.setType(valueType);  
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");  
physicalElement = addElement(physicalInterface,"ElectricalElement",...  
    Type="electrical.electrical");  
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.

Interfaces			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

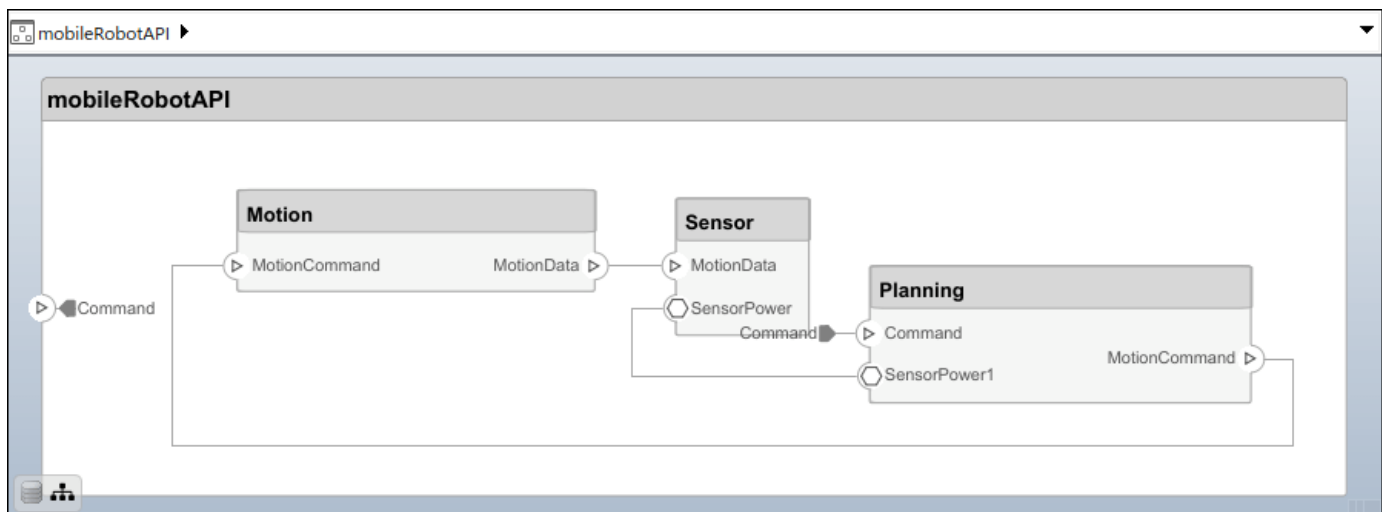
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```





## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double', Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

### Add Hierarchy

Add two components named Controller and Scope inside the Motion component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);
```

For outport connections, the data element must be specified.

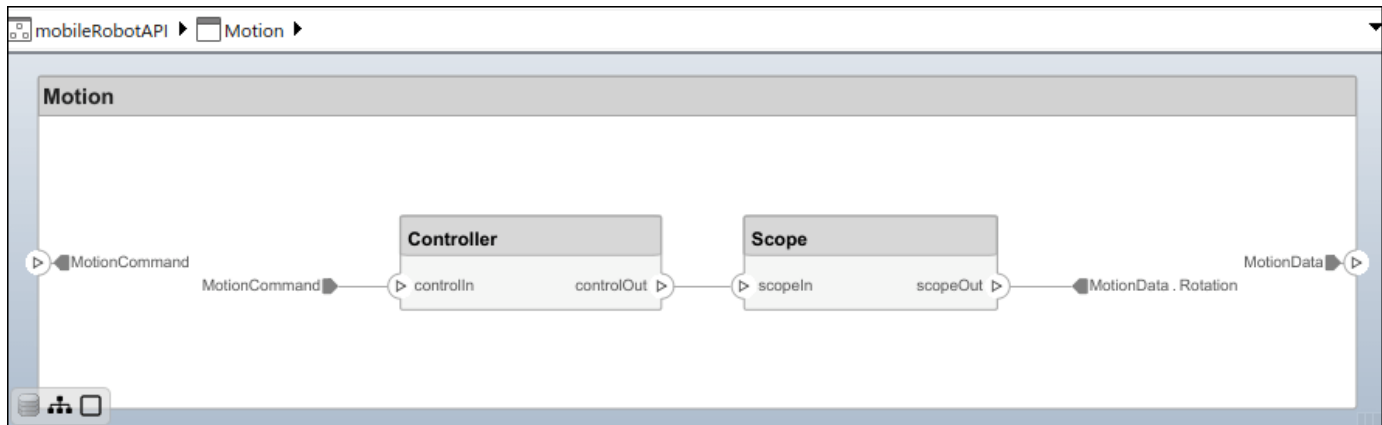
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save
```

```
linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the `Planning` component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

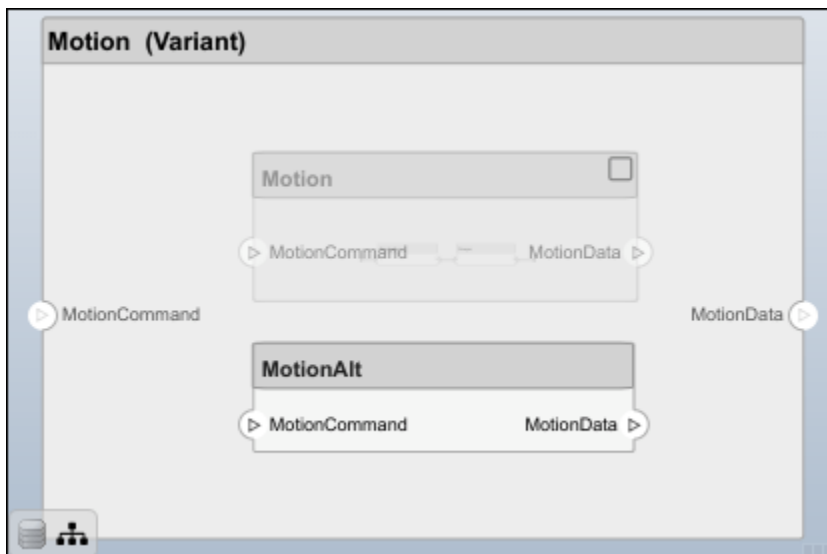
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

## Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

[createInterface](#) | [addPhysicalInterface](#) | [systemcomposer.interface.PhysicalInterface](#) | [systemcomposer.interface.PhysicalElement](#) | [systemcomposer.interface.Dictionary](#)

### Topics

[“Specify Physical Interfaces on Ports”](#)  
[“Create Interfaces”](#)  
[“Manage Interfaces with Data Dictionaries”](#)

# systemcomposer.interface.PhysicalElement

Physical element in physical interface

## Description

A `PhysicalElement` object represents a physical element in a physical interface.

## Creation

Create a physical element using the `addElement` function.

```
element = addElement(interface, "newPhysicalElement")
```

## Properties

### Interface — Parent physical interface of physical element

physical interface object

Parent physical interface of physical element, specified as a `systemcomposer.interface.PhysicalInterface` object.

### Name — Physical element name

character vector | string

Physical element name, specified as a character vector or string.

Example: "newPhysicalElement"

Data Types: char | string

### Type — Type of physical element

physical interface object | physical domain object | character vector | string

Type of physical element, specified as a `systemcomposer.interface.PhysicalInterface` or `systemcomposer.interface.PhysicalDomain` object or a character vector or string of the partial physical domain name. For a list of valid physical domain names, see “Domain-Specific Line Styles” (Simscape).

### UUID — Universal unique identifier

character vector

Universal unique identifier for physical element, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### ExternalUUID — Unique external identifier

character vector



Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the physical element and through all operations that preserve the UUID.

Data Types: char

## Object Functions

setName Set name for value type, function argument, interface, or element  
destroy Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.

	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

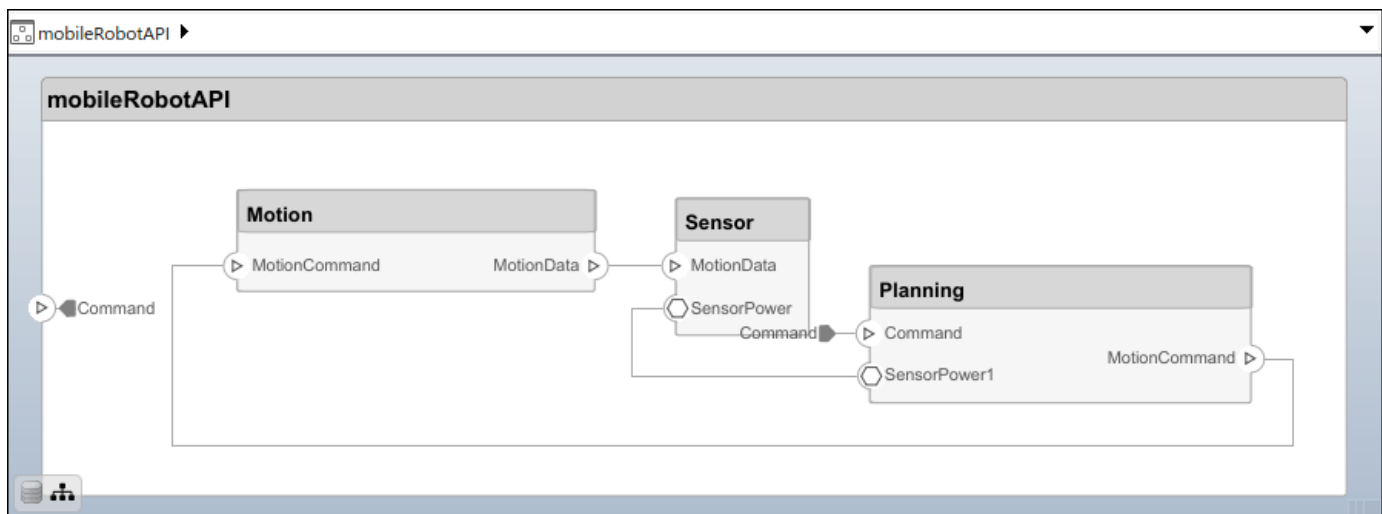
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile, "projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile, "physicalComponent", AppliesTo="Component");  
sCompSType = addStereotype(profile, "softwareComponent", AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile, "standardConn", AppliesTo="Connector");
```

### **Add Properties**

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");  
addProperty(elemSType, 'Description', Type="string");  
addProperty(pCompSType, 'Cost', Type="double", Units="USD");  
addProperty(pCompSType, 'Weight', Type="double", Units="g");  
addProperty(sCompSType, 'develCost', Type="double", Units="USD");  
addProperty(sCompSType, 'develTime', Type="double", Units="hour");  
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");  
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");  
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### **Save Profile**

```
profile.save;
```

### **Apply Profile to Model**

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")  
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")  
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");  
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');  
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
```

```

    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

### Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```

motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);

```

For outport connections, the data element must be specified.

```

c_planningScope = connect(scopeCompPortOut, motionPorts(2), DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, ...
    "GeneralProfile.standardConn");

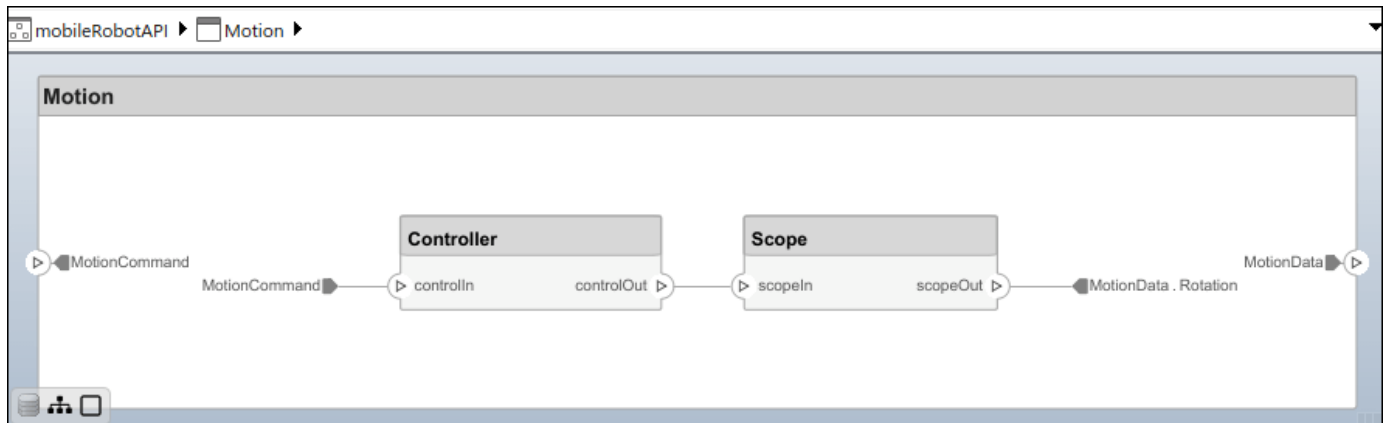
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the **Controller** component into a reference component to reference the new model. To add additional ports on the **Controller** component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "Gyroscope");
referenceModel.save
```

```
linkToModel(motionController, "mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the **Planning** component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active

choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

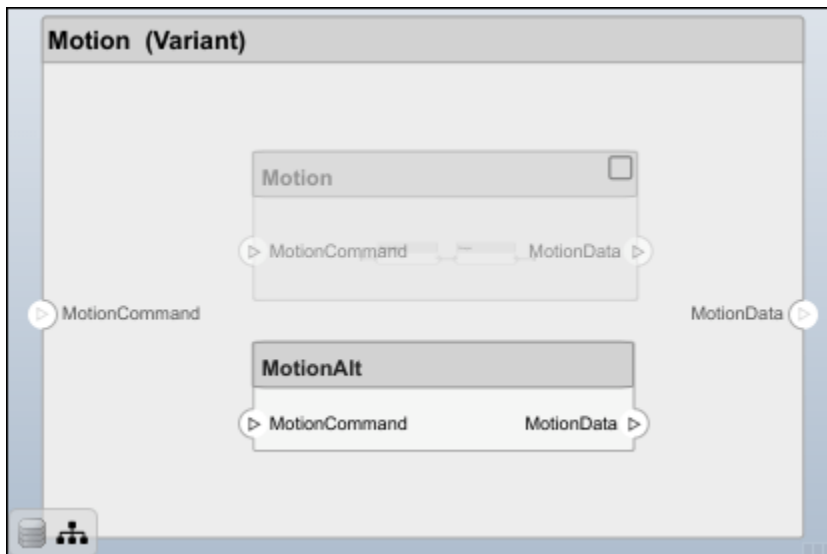
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

cleanUpArtifacts

## More About

### Definitions

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"



Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`addPhysicalInterface` | `addElement` | `removeElement` | `getElement` | `systemcomposer.interface.Dictionary` | `systemcomposer.interface.PhysicalDomain` | `systemcomposer.interface.PhysicalInterface`

### Topics

“Specify Physical Interfaces on Ports”  
“Create Interfaces”  
“Manage Interfaces with Data Dictionaries”

# systemcomposer.interface.PhysicalInterface

Physical interface

## Description

A `PhysicalInterface` object represents the structure of a physical interface.

## Creation

Create a physical interface using the `addPhysicalInterface` function.

```
interface = addPhysicalInterface(model.InterfaceDictionary, "newPhysicalInterface")
```

## Properties

### Owner — Parent of physical interface

dictionary object | physical element object | architecture port object

Parent of physical interface, specified as a `systemcomposer.interface.Dictionary`, `systemcomposer.interface.PhysicalElement`, or `systemcomposer.arch.ArchitecturePort` object.

### Model — Parent model

model object

Parent System Composer model of physical interface, specified as a `systemcomposer.arch.Model` object.

### Name — Physical interface name

character vector | string

Physical interface name, specified as a character vector or string. This property must be a valid MATLAB identifier.

Example: "newPhysicalInterface"

Data Types: char | string

### Elements — Elements in interface

array of physical element objects

Elements in interface, specified as an array of `systemcomposer.interface.PhysicalElement` objects.

### Description — Physical interface description

character vector | string

Physical interface description, specified as a character vector or string.

Data Types: char | string

**UUID – Universal unique identifier**

character vector

Universal unique identifier for physical interface, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

**ExternalUID – Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the physical interface and through all operations that preserve the UUID.

Data Types: char

**Object Functions**

addElement	Add element
getElement	Get object for element
removeElement	Remove element
setName	Set name for value type, function argument, interface, or element
setDescription	Set description for value type or interface
applyStereotype	Apply stereotype to architecture model element
getStereotypes	Get stereotypes applied on element of architecture model
getStereotypeProperties	Get stereotype property names on element
removeStereotype	Remove stereotype from model element
getProperty	Get property value corresponding to stereotype applied to element
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from element
setProperty	Set property value corresponding to stereotype applied to element
hasStereotype	Find if element has stereotype applied
hasProperty	Find if element has property
destroy	Remove model element

**Examples****Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

**Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

**Add Components, Ports, Connections, and Interfaces**

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

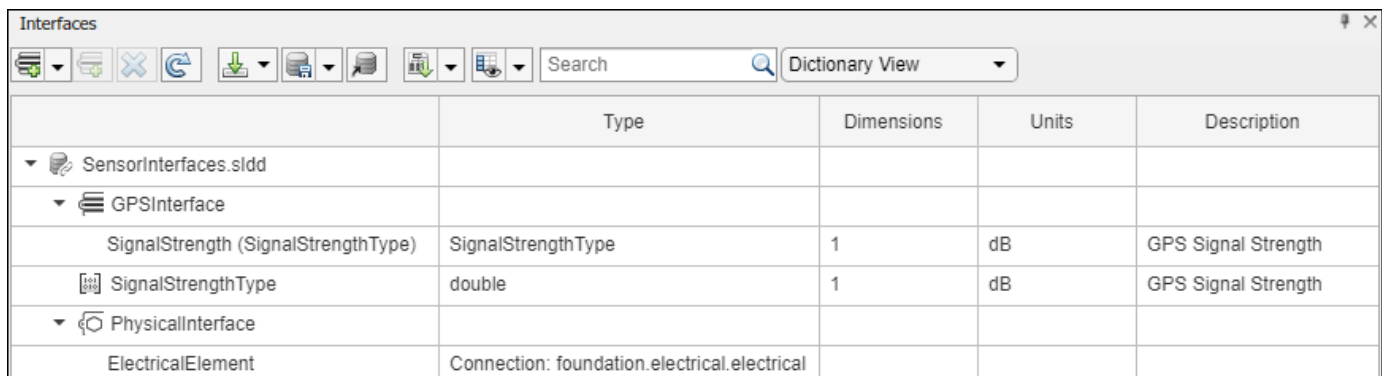
Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.



The screenshot shows the 'Interfaces' window with a toolbar and a table. The table has columns for Type, Dimensions, Units, and Description. It lists the following elements:

	Type	Dimensions	Units	Description
▼ SensorInterfaces.slidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
```

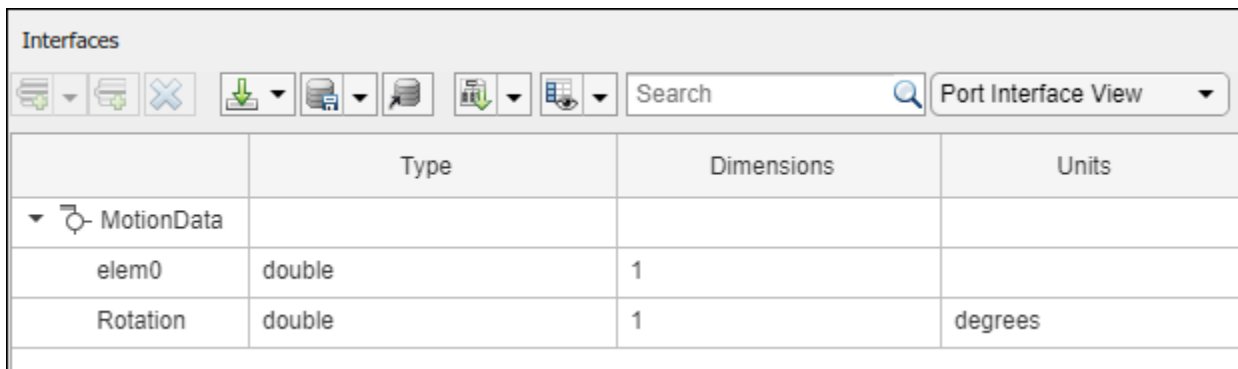
```
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, ...
    {'in', 'out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces			
<span>Icons</span> <input type="text" value="Search"/> <span>Port Interface View</span>			
	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch, componentSensor, componentPlanning, Rule="interface");
c_motionData = connect(arch, componentMotion, componentSensor);
c_motionCommand = connect(arch, componentPlanning, componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

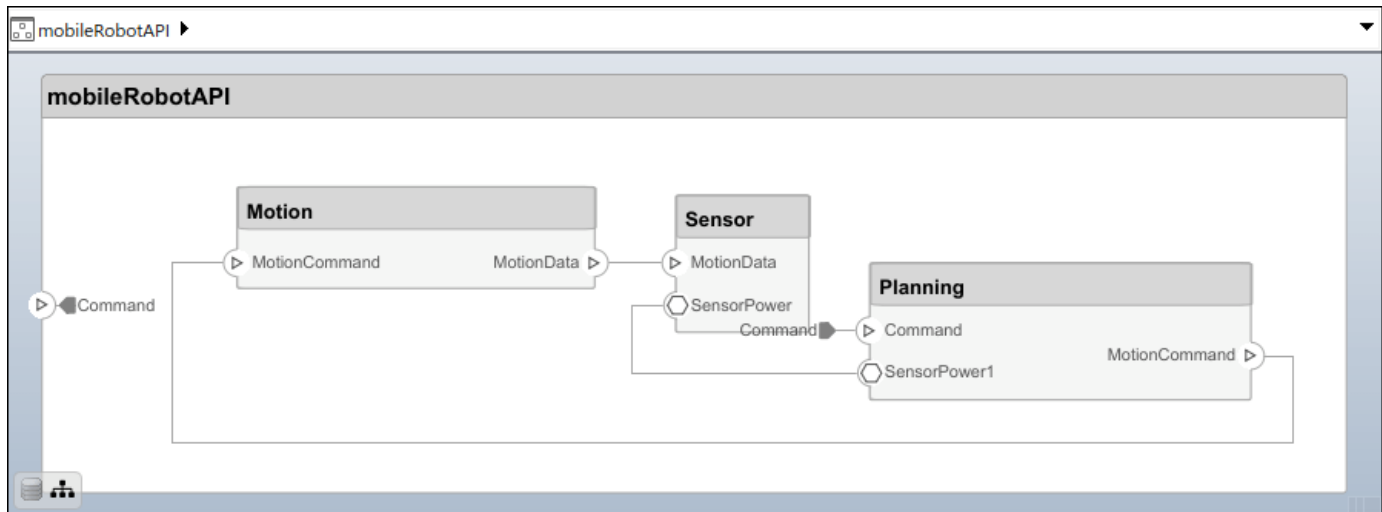
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType,'ID',Type="uint8");
addProperty(elemSType,'Description',Type="string");
addProperty(pCompSType,'Cost',Type="double",Units="USD");
addProperty(pCompSType,'Weight',Type="double",Units="g");
addProperty(sCompSType,'develCost',Type="double",Units="USD");
addProperty(sCompSType,'develTime',Type="double",Units="hour");
```

```
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```



## Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

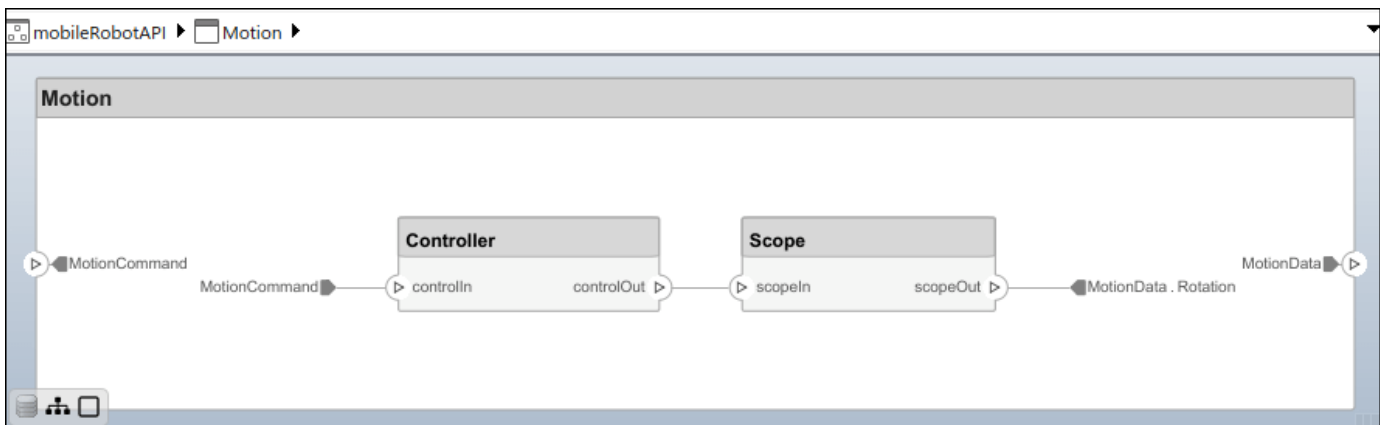
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### **Make Variant Component**

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

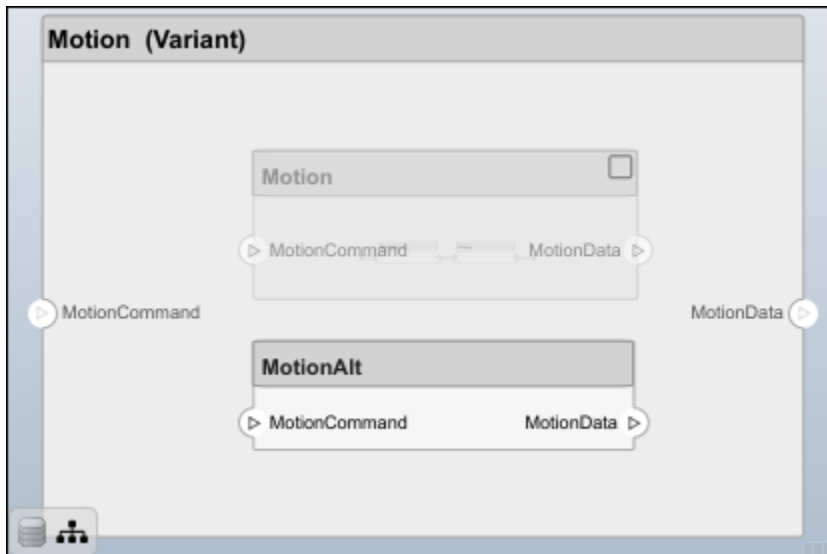
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanupArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"

Term	Definition	Application	More Information
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>

Term	Definition	Application	More Information
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

[addPhysicalInterface](#) | [setInterface](#) | [getInterface](#) | [getInterfaceNames](#) | [removeInterface](#) | [systemcomposer.interface.Dictionary](#) | [systemcomposer.interface.PhysicalElement](#) | [systemcomposer.interface.PhysicalDomain](#)

### Topics

[“Specify Physical Interfaces on Ports”](#)  
[“Create Interfaces”](#)  
[“Manage Interfaces with Data Dictionaries”](#)

# systemcomposer.interface.ServiceInterface

Client-server interface

## Description

A `ServiceInterface` object describes the structure and attributes of a client-server interface.

## Creation

Create a service interface using the `addServiceInterface` function.

```
interface = addServiceInterface(model.InterfaceDictionary, "newServiceInterface")
```

## Properties

### Dictionary — Dictionary of service interface

dictionary object

Dictionary of service interface, specified as a `systemcomposer.interface.Dictionary` object.

### Model — Parent model

model object

Parent model of service interface, specified as a `systemcomposer.arch.Model` object.

### Name — Service interface name

character vector | string

Service interface name, specified as a character vector or string. This property must be a valid MATLAB identifier.

Example: "newInterface"

Data Types: char | string

### Elements — Elements in interface

array of function element objects

Elements in interface, specified as an array of `systemcomposer.interface.FunctionElement` objects.

### Description — Service interface description

character vector | string

Service interface description, specified as a character vector or string.

Data Types: char | string

### UUID — Universal unique identifier

character vector

Universal unique identifier for service interface, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### **ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the service interface and through all operations that preserve the UUID.

Data Types: char

## **Object Functions**

<code>addElement</code>	Add element
<code>getElement</code>	Get object for element
<code>removeElement</code>	Remove element
<code>setName</code>	Set name for value type, function argument, interface, or element
<code>setDescription</code>	Set description for value type or interface
<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>removeStereotype</code>	Remove stereotype from model element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>destroy</code>	Remove model element

## **Examples**

### **Get Function Argument**

Create a new model.

```
model = systemcomposer.createModel("archModel", "SoftwareArchitecture", true);
```

Create a service interface.

```
interface = addServiceInterface(model.InterfaceDictionary, "newServiceInterface");
```

Create a function element.

```
element = addElement(interface, "newFunctionElement");
```

Set a function prototype to add function arguments.

```
setFunctionPrototype(element, "y=f0(u)")
```

Get a function argument.

```
argument = getFunctionArgument(element, "y")
```



argument =

FunctionArgument with properties:

```

Interface: [1x1 systemcomposer.interface.ServiceInterface]
Element: [1x1 systemcomposer.interface.FunctionElement]
  Name: 'y'
  Type: [1x1 systemcomposer.ValueType]
Dimensions: '1'
Description: ''
  UUID: '018b4e55-fa8f-4250-ac2b-df72bf620feb'
ExternalUID: ''

```

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”

Term	Definition	Application	More Information
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution — When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> <li>• Asynchronous execution — When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul>	<code>systemcomposer.interface.FunctionElement</code>
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	<code>systemcomposer.interface.FunctionArgument</code>

Term	Definition	Application	More Information
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	"Class Diagram View of Software Architectures"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• "Create Architecture Model with Interfaces and Requirement Links"</li> <li>• "Define Port Interfaces Between Components"</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• "Create Interfaces"</li> <li>• "Assign Interfaces to Ports"</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	"Create Value Types as Interfaces"
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	"Define Owned Interfaces Local to Ports"
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the "Interface Adapter" dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• "Interface Adapter"</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2022a

## **See Also**

setInterface | getInterface | getInterfaceNames | removeInterface |  
systemcomposer.interface.FunctionElement | systemcomposer.interface.Dictionary

## **Topics**

“Author Service Interfaces for Client-Server Communication”  
“Client-Server Interfaces in Class Diagram View”  
“Define Port Interfaces Between Components”

## systemcomposer.interface.SignalElement

(Removed) Element in signal interface

---

**Note** The `systemcomposer.interface.SignalElement` class has been removed. It has been replaced with the `systemcomposer.interface.DataElement` class. For further details, see “Compatibility Considerations”.

---

### Description

A `SignalElement` object represents a signal element in a signal interface.

### Properties

#### Interface — Parent interface of element

signal interface object

Parent interface of element, specified as a `systemcomposer.interface.SignalInterface` object.

#### Name — Element name

character vector

Element name, specified as a character vector.

Data Types: `char`

#### Dimensions — Dimensions of element

array of positive integers

Dimensions of element, specified as an array of positive integers.

Data Types: `integer`

#### Type — Data type of element

character vector

Data type of element, specified as a character vector.

Data Types: `char`

#### Complexity — Complexity of element

'real' | 'complex'

Complexity of element, specified as 'real' or 'complex'.

Data Types: `char`

#### Units — Units of element

character vector

Units of element, specified as a character vector.

Data Types: char

### **Minimum — Minimum value for element**

numeric

Minimum value for element, specified as a numeric double.

Data Types: double

### **Maximum — Maximum value for element**

numeric

Maximum value for element, specified as a numeric double.

Data Types: double

### **Description — Description text for element**

character vector

Description text for element, specified as a character vector.

Data Types: char

### **UUID — Universal unique identifier**

character vector

Universal unique identifier for interface element, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### **ExternalUUID — Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the interface element and through all operations that preserve the UUID.

Data Types: char

## **Object Functions**

setName	Set name for value type, function argument, interface, or element
setDataType	Set data type for value type
setDimensions	Set dimensions for value type
setUnits	Set units for value type
setComplexity	Set complexity for value type
setMinimum	Set minimum for value type
setMaximum	Set maximum for value type
setDescription	Set description for value type or interface
destroy	Remove model element

## **Version History**

**Introduced in R2019a**

**R2021b: systemcomposer.interface.SignalElement class has been removed**

*Errors starting in R2021b*

The `systemcomposer.interface.SignalElement` class is removed in R2021b. Use `systemcomposer.interface.DataElement` instead.

**See Also**

`systemcomposer.interface.DataInterface` | `systemcomposer.interface.DataElement` | `systemcomposer.interface.Dictionary` | `systemcomposer.ValueType` | `addElement` | `removeElement` | `getElement`

**Topics**

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”



# systemcomposer.interface.SignalInterface

(Removed) Signal interface

---

**Note** The `systemcomposer.interface.SignalInterface` class has been removed. It has been replaced with the `systemcomposer.interface.DataInterface` class. For further details, see “Compatibility Considerations”.

---

## Description

A `SignalInterface` object represents the structure of the signal interface at a given port.

## Properties

### Dictionary — Parent dictionary of interface

interface dictionary object

Parent dictionary of interface, specified as a `systemcomposer.interface.Dictionary` object.

### Name — Interface name

character vector

Interface name, specified as a character vector.

Example: `'NewInterface'`

Data Types: `char`

### Elements — Elements in interface

array of interface element objects

Elements in interface, specified as an array of `systemcomposer.interface.SignalElement` objects.

### UUID — Universal unique identifier

character vector

Universal unique identifier for signal interface, specified as a character vector.

Example: `'91d5de2c-b14c-4c76-a5d6-5dd0037c52df'`

Data Types: `char`

### ExternalUUID — Unique external identifier

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the signal interface and through all operations that preserve the **UUID**.

Data Types: `char`

### Model — Parent model

model object

Parent System Composer model of signal interface, specified as a `systemcomposer.arch.Model` object.

## Object Functions

<code>addElement</code>	Add element
<code>getElement</code>	Get object for element
<code>removeElement</code>	Remove element
<code>applyStereotype</code>	Apply stereotype to architecture model element
<code>getStereotypes</code>	Get stereotypes applied on element of architecture model
<code>getStereotypeProperties</code>	Get stereotype property names on element
<code>removeStereotype</code>	Remove stereotype from model element
<code>getProperty</code>	Get property value corresponding to stereotype applied to element
<code>getPropertyValue</code>	Get value of architecture property
<code>getEvaluatedPropertyValue</code>	Get evaluated value of property from element
<code>setProperty</code>	Set property value corresponding to stereotype applied to element
<code>hasStereotype</code>	Find if element has stereotype applied
<code>hasProperty</code>	Find if element has property
<code>destroy</code>	Remove model element

## Version History

### Introduced in R2019a

### **R2021b: `systemcomposer.interface.SignalInterface` class has been removed**

*Errors starting in R2021b*

The `systemcomposer.interface.SignalInterface` class is removed in R2021b. Use `systemcomposer.interface.DataInterface` instead.

## See Also

`systemcomposer.interface.DataInterface` | `systemcomposer.interface.DataElement` | `systemcomposer.interface.Dictionary` | `systemcomposer.ValueType` | `addInterface` | `getInterface` | `removeInterface` | `getInterfaceNames`

## Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

# systemcomposer.io.ModelBuilder

Model builder for System Composer architecture models

## Description

Build System Composer models using a `systemcomposer.io.ModelBuilder` object. Build System Composer models with components and their position in an architecture hierarchy, ports and their mappings to components, connections among components through ports, and interfaces in architecture models and their mappings to ports.

## Creation

```
builder = systemcomposer.io.ModelBuilder(profile)
```

## Properties

### Components — Component information

table

Component information, specified as a table containing this information:

- Hierarchical information of components
- Type of component (for example, Component, Reference Component, Variant Component, or Adapter)
- Stereotypes applied on a component
- Ability to set property values of a component

### Ports — Ports information

table

Ports information, specified as a table. The table contains the information about ports, including their mappings to components and interfaces, and stereotypes applied on them.

### Connections — Connections information

table

Connections information, specified as a table. The table contains information about the connections between the ports defined in Ports table as well as stereotypes applied on connections.

### Interfaces — Interfaces information

table

Interfaces information, specified as a table. The table contains the definitions of various interfaces and their elements.

## Examples

## Import System Composer Architecture Using ModelBuilder Class

Import architecture specifications into System Composer™ using the `systemcomposer.io.ModelBuilder` utility class. These architecture specifications can be defined in an external source, such as an Excel® file.

In System Composer, an architecture is fully defined by four sets of information:

- Components and their position in the architecture hierarchy.
- Ports and their mapping to components.
- Connections among components through ports. In this example, we also import interface data definitions from an external source.
- Interfaces in architecture models and their mapping to ports.

This example uses the `systemcomposer.io.ModelBuilder` class to pass all of the above architecture information and import a System Composer model.

In this example, architecture information of a small UAV system is defined in an Excel spreadsheet and is used to create a System Composer architecture model.

### External Source Files

- `Architecture.xlsx` — This Excel file contains hierarchical information of the architecture model. This example maps the external source data to System Composer model elements. This information maps in column names to System Composer model elements.

```
# Element      : Name of the element. Either can be component or port name.
# Parent       : Name of the parent element.
# Class        : Can be either component or port(Input/Output direction of the port).
# Domain       : Mapped as component property. Property "Manufacturer" defined in the
                 profile UAVComponent under Stereotype PartDescriptor maps to Domain values in
# Kind         : Mapped as component property. Property "ModelName" defined in the
                 profile UAVComponent under Stereotype PartDescriptor maps to Kind values in
# InterfaceName : If class is of port type. InterfaceName maps to name of the interface link
# ConnectedTo  : In case of port type, it specifies the connection to
                 other port defined in format "ComponentName::PortName".
```

- `DataDefinitions.xlsx` — This Excel file contains interface data definitions of the model. This example assumes this mapping between the data definitions in the Excel source file and interfaces hierarchy in System Composer.

```
# Name         : Name of the interface or element.
# Parent       : Name of the parent interface Name(Applicable only for elements) .
# Datatype     : Datatype of element. Can be another interface in format
                 Bus: InterfaceName
# Dimensions   : Dimensions of the element.
# Units        : Unit property of the element.
# Minimum      : Minimum value of the element.
# Maximum      : Maximum value of the element.
```

### Step 1. Instantiate the ModelBuilder Class

You can instantiate the `ModelBuilder` class with a profile name.

```
[stat,fa] = fileattrib(pwd);
if ~fa.UserWrite
```

```

        disp('This script must be run in a writable directory');
        return;
    end

```

Specify the name of the model to build.

```
modelName = 'scExampleModelBuilder';
```

Specify the name of the profile.

```
profile = 'UAVComponent';
```

Specify the name of the source file to read architecture information.

```
architectureFileName = 'Architecture.xlsx';
```

Instantiate the ModelBuilder.

```
builder = systemcomposer.io.ModelBuilder(profile);
```

## Step 2. Build Interface Data Definitions

Reading the information in the external source file `DataDefinitions.xlsx` to build the interface data model.

Create MATLAB® tables from the Excel source file.

```
opts = detectImportOptions('DataDefinitions.xlsx');
opts.DataRange = 'A2';
```

Force readtable to start reading from the second row.

```
definitionContents = readtable('DataDefinitions.xlsx',opts);
```

The `systemcomposer.io.IdService` class generates unique ID for a given key.

```
idService = systemcomposer.io.IdService();
for rowItr = 1:numel(definitionContents(:,1))
    parentInterface = definitionContents.Parent{rowItr};
    if isempty(parentInterface)

```

In the case of interfaces, add the interface name to the model builder.

```
        interfaceName = definitionContents.Name{rowItr};
```

Get the unique interface ID.

`getID(container, key)` generates or returns (if key is already present) same value for input key within the container.

```
        interfaceID = idService.getID('interfaces',interfaceName);
```

Use `builder.addInterface` to add the interface to the data dictionary.

```
        builder.addInterface(interfaceName,interfaceID);
    else

```

In the case of an element, read the element properties and add the element to the parent interface.

```

elementName = definitionContents.Name{rowItr};
interfaceID = idService.getID('interfaces',parentInterface);

```

The ElementID is unique within a interface. Append E at the start of an ID for uniformity. The generated ID for an input element is unique within parent interface name as a container.

```

elemID = idService.getID(parentInterface,elementName,'E');

```

Set the data type, dimensions, units, minimum, and maximum properties of the element.

```

datatype = definitionContents.DataType{rowItr};
dimensions = string(definitionContents.Dimensions(rowItr));
units = definitionContents.Units(rowItr);

```

Make sure that input to builder utility function is always a string.

```

if ~ischar(units)
    units = '';
end
minimum = definitionContents.Minimum{rowItr};
maximum = definitionContents.Maximum{rowItr};

```

Use builder.addElementInInterface to add an element with properties in the interface.

```

    builder.addElementInInterface(elementName,elemID,interfaceID,datatype,dimensions,units,'E');
end
end

```

### Step 3. Build Architecture Specifications

Architecture specifications are created by MATLAB tables from the Excel source file.

```

excelContents = readtable(architectureFileName);

```

Iterate over each row in the table.

```

for rowItr =1:numel(excelContents(:,1))

```

Read each row of the Excel file and columns.

```

    class = excelContents.Class(rowItr);
    Parent = excelContents.Parent(rowItr);
    Name = excelContents.Element{rowItr};

```

Populate the contents of the table.

```

    if strcmp(class,'component')
        ID = idService.getID('comp',Name);

```

The Root ID is by default set as zero.

```

        if strcmp(Parent,'scExampleSmallUAV')
            parentID = "0";
        else
            parentID = idService.getID('comp',Parent);
        end

```

Use builder.addComponent to add a component.

```

        builder.addComponent(Name,ID,parentID);

```

Read the property values.

```
kind = excelContents.Kind{rowItr};
domain = excelContents.Domain{rowItr};
```

Use `builder.setComponentProperty` to set stereotype and property values.

```
builder.setComponentProperty(ID, 'StereotypeName', 'UAVComponent.PartDescriptor', 'ModelName');
else
```

In this example, concatenation of the port name and parent component name is used as key to generate unique IDs for ports.

```
portID = idService.getID('port',strcat(Name,Parent));
```

For ports on root architecture, the `compID` is assumed as 0.

```
if strcmp(Parent, 'scExampleSmallUAV')
    compID = "0";
else
    compID = idService.getID('comp',Parent);
end
```

Use `builder.addPort` to add a port.

```
builder.addPort(Name,class,portID,compID );
```

The `InterfaceName` specifies the name of the interface linked to the port.

```
interfaceName = excelContents.InterfaceName{rowItr};
```

Get the interface ID.

`getID` will return the same IDs already generated while adding interface in Step 2.

```
interfaceID = idService.getID('interfaces',interfaceName);
```

Use `builder.addInterfaceToPort` to map interface to port.

```
builder.addInterfaceToPort(interfaceID,portID);
```

Read the `ConnectedTo` information to build connections between components.

```
connectedTo = excelContents.ConnectedTo{rowItr};
```

`ConnectedTo` is in the format:

```
(DestinationComponentName::DestinationPortName)
```

For this example, consider the current port as source of the connection.

```
if ~isempty(connectedTo)
    connID = idService.getID('connection',connectedTo);
    splits = split(connectedTo, '::');
```

Get the port ID of the connected port.

In this example, port ID is generated by concatenating the port name and the parent component name. If the port ID is already generated, the `getID` function returns the same ID for the input key.

```
connectedPortID = idService.getID('port',strcat(splits(2),splits(1)));
```

Populate the connection table.

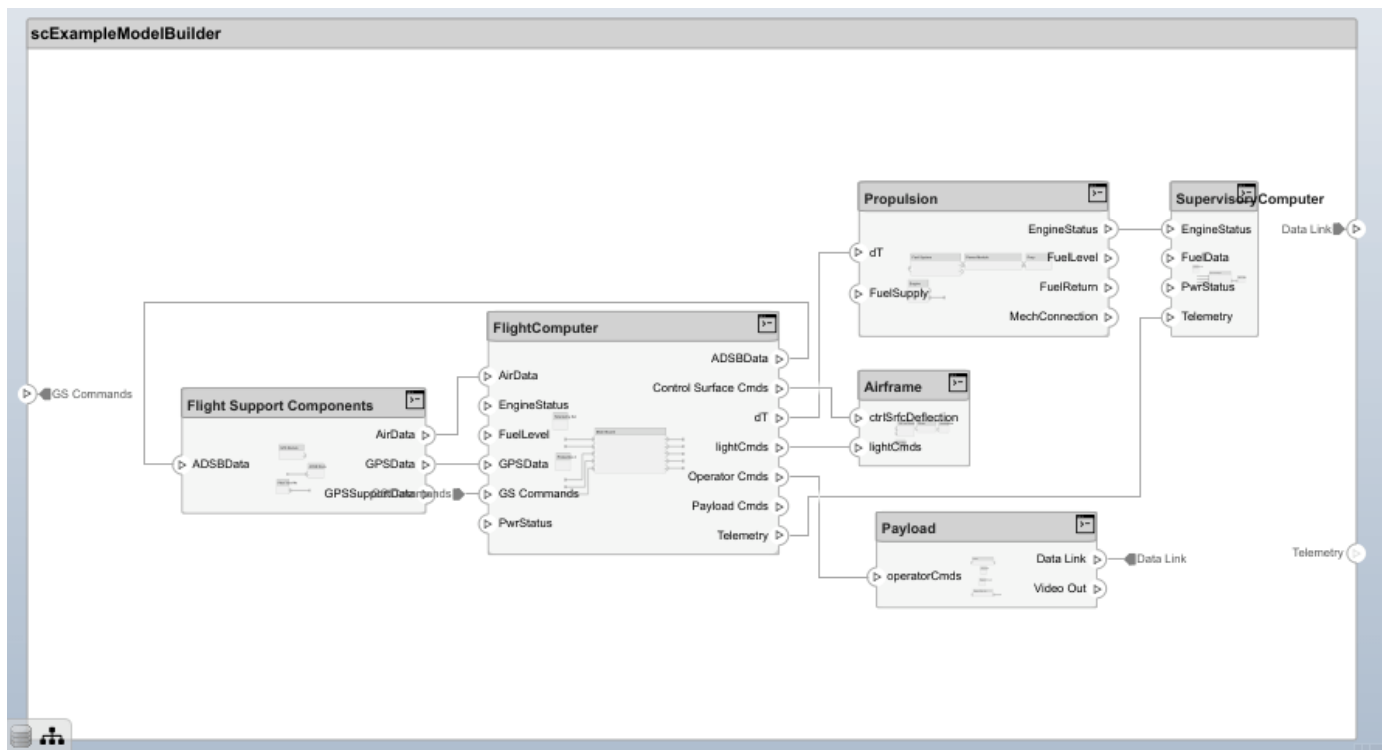
```
sourcePortID = portID;
destPortID = connectedPortID;
```

Use builder.addConnection to add connections.

```
builder.addConnection(connectedTo,connID,sourcePortID,destPortID);
end
end
end
```

### Step 3. Import Model from Populated Tables with builder.build Function

```
[model,importReport] = builder.build(modelName);
```



Clean up artifacts.

```
cleanup
```



Copyright 2020 The MathWorks, Inc.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## Algorithms

Components	Description
<code>addComponent(compName, ID, ParentID)</code>	Add component with name and ID as a child of component with ID as ParentID. In case of root, ParentID is 0.
<code>setComponentProperty(ID, varargin)</code>	Set stereotype on component with ID. Key value pair of property name and value defined in the stereotype can be passed as input. In this example  <pre>builder.setComponentProperty(ID, 'StereotypeName', ... 'UAVComponent.PartDescriptor', 'ModelName', kind, 'Manufacturer', domain)</pre> ModelName and Manufacturer are properties under stereotype PartDescriptor.

Ports	Description
<code>addPort(portName, direction, ID, compID)</code>	Add port with name and ID with direction (either Input or Output) to component with ID as compID.
<code>setPropertyOnPort(ID, varargin)</code>	Set stereotype on port with ID. Key value pair of the property name and the value defined in the stereotype can be passed as input.

Connections	Description
<code>addConnection(connName, ID, sourcePortID, destPortID)</code>	Add connection with name and ID between ports with <code>sourcePortID</code> (direction: Output) and <code>destPortID</code> (direction: Input) defined in the ports table.
<code>setPropertyOnConnection(ID, varargin)</code>	Set stereotype on connection with ID. Key value pair of the property name and the value defined in the stereotype can be passed as input.

Interfaces	Description
<code>addInterface(interfaceName, ID)</code>	Add interface with name and ID to a data dictionary.
<code>addElementInInterface(elementName, ID, interfaceID, datatype, dimensions, units, complexity, Maximum, Minimum)</code>	Add element with name and ID under an interface with ID as <code>interfaceID</code> . Data types, dimensions, units, complexity, and maximum and minimum are properties of an element. These properties are specified as strings.
<code>addAnonymousInterface(ID, datatype, dimensions, units, complexity, Maximum, Minimum)</code>	Add anonymous interface with ID and element properties like data type, dimensions, units, complexity, maximum, and minimum. Data type of an owned interface cannot be another interface name. Owned interfaces do not have elements like other interfaces.

Interfaces and Ports	Description
<code>addInterfaceToPort(interfaceID, portID)</code>	Link an interface with ID specified as <code>InterfaceID</code> to a port with ID specified as <code>PortID</code> .

Models	Description
<code>build(modelName)</code>	Build model with model name passed as input.

Logging and Reporting	Description
<code>getImportErrorLog</code>	Get <code>ErrorLogs</code> generated while importing the model. Called after the <code>build</code> function.
<code>getImportReport</code>	Get a report of the import. Called after the <code>build</code> function.

## Version History

Introduced in R2019b

### See Also

`importModel` | `exportModel`

**Topics**

“Import and Export Architecture Models”

# systemcomposer.parameter.ParameterDefinition

(Not recommended) Parameter definition in System Composer

---

**Note** The `systemcomposer.parameter.ParameterDefinition` object is not recommended. Use the `systemcomposer.arch.Parameter` object instead. For more information, see “Compatibility Considerations”.

---

## Description

A `ParameterDefinition` object describes a parameter definition in System Composer. Set and get the properties of a parameter definition to edit and view the instance-specific parameters specified as model arguments on a referenced model.

## Creation

Creating a `ParameterDefinition` object directly is not supported. A `ParameterDefinition` object is returned when you use the `getParameterDefinition` function.

## Properties

### Owner — Element that owns definition

architecture object

Element that owns definition, specified as a `systemcomposer.arch.Architecture` object.

### Name — Parameter name

character vector | string

Parameter name, specified as a character vector or string. This property must be a valid MATLAB identifier.

Example: "AirSpeed"

Data Types: char | string

### Type — Parameter data type

character vector | string

Parameter data type, specified as a character vector or string. This property must be a valid MATLAB data type.

Data Types: char | string

### Dimensions — Parameter dimensions

character vector | string

Parameter dimensions, specified as a character vector or string.

Data Types: char | string

**Unit – Parameter units**

character vector | string

Parameter units, specified as a character vector or string.

Data Types: char | string

**Min – Parameter minimum**

character vector | string

Parameter minimum, specified as a character vector or string.

Data Types: char | string

**Max – Parameter maximum**

character vector | string

Parameter maximum, specified as a character vector or string.

Data Types: char | string

## Version History

**Introduced in R2022a**

**R2022b\_plus: systemcomposer.parameter.ParameterDefinition object is not recommended**

*Not recommended starting in R2022b\_plus*

The `systemcomposer.parameter.ParameterDefinition` object is not recommended. Use the `systemcomposer.arch.Parameter` object instead.

**See Also**

`getEvaluatedParameterValue` | `getParameterDefinition` | `getParameterNames` | `getParameterValue` | `setParameterValue` | `setUnit`

**Topics**

“Access Model Arguments as Parameters on Reference Components”

“Use Parameters to Store Instance Values with Components”

# systemcomposer.profile.Profile

Profile

## Description

A Profile object represents a profile for a System Composer model.

## Creation

Create a profile using the `systemcomposer.profile.Profile.createProfile` function.

```
profile = systemcomposer.profile.Profile.createProfile("profileName");
```

---

**Note** Before you move, copy, or rename a profile to a different directory, you must close the profile in the **Profile Editor** or by using the `close` function. If you rename a profile, follow the example for the `renameProfile` function.

---

## Properties

### Name — Name of profile

character vector | string

Name of profile, specified as a character vector or string. This property must be a valid MATLAB identifier.

Data Types: char | string

### FriendlyName — Descriptive name of profile

character vector | string

Descriptive name of profile, specified as a character vector or string. This property can contain spaces and special characters, but no new lines.

Data Types: char | string

### Description — Description text for profile

multi-line character vector | multi-line string

Description text for profile, specified as a multi-line character vector or string.

Data Types: char | string

### Stereotypes — Stereotypes

array of stereotype objects

Stereotypes defined in profile, specified as an array of `systemcomposer.profile.Stereotype` objects.

Data Types: char

## Object Functions

createProfile	Create profile
addStereotype	Add stereotype to profile
removeStereotype	Remove stereotype from profile
getStereotype	Find stereotype in profile by name
getDefaultStereotype	Get default stereotype for profile
setDefaultStereotype	Set default stereotype for profile
find	Find profile by name
open	Open profile
load	Load profile from file
save	Save profile as file
close	Close profile
closeAll	Close all open profiles
destroy	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.



```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.slidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData', 'SensorPower'},...
    {'in', 'physical'});
sensorPorts(2).setInterface(physicalInterface)
```



```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command', 'SensorPower1', 'MotionCommand'},...
    {'in', 'physical', 'out'});
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand', 'MotionData'},...
    {'in', 'out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.

Interfaces			
 <input type="text" value="Search"/> <span>Port Interface View</span>			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

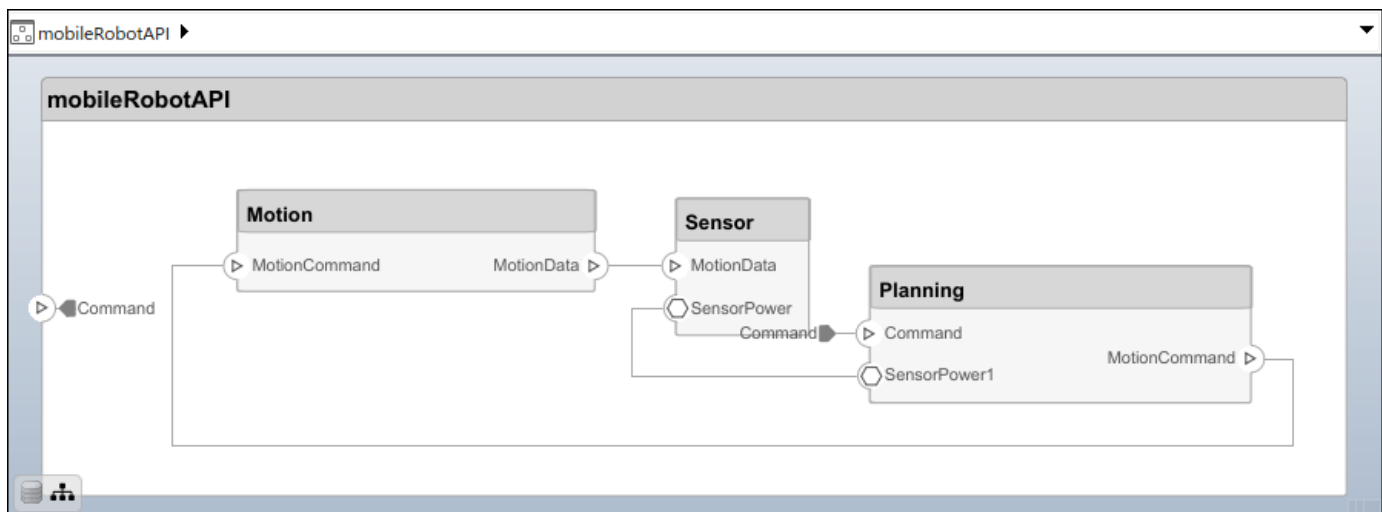
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double', Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

### Add Hierarchy

Add two components named Controller and Scope inside the Motion component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);
```

For outport connections, the data element must be specified.

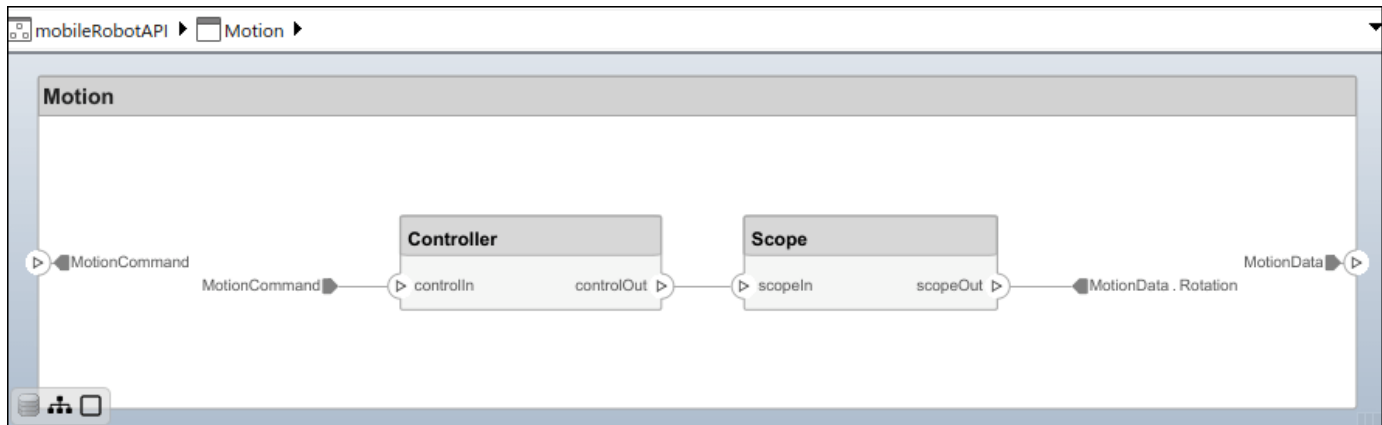
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save
```

```
linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the `Planning` component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

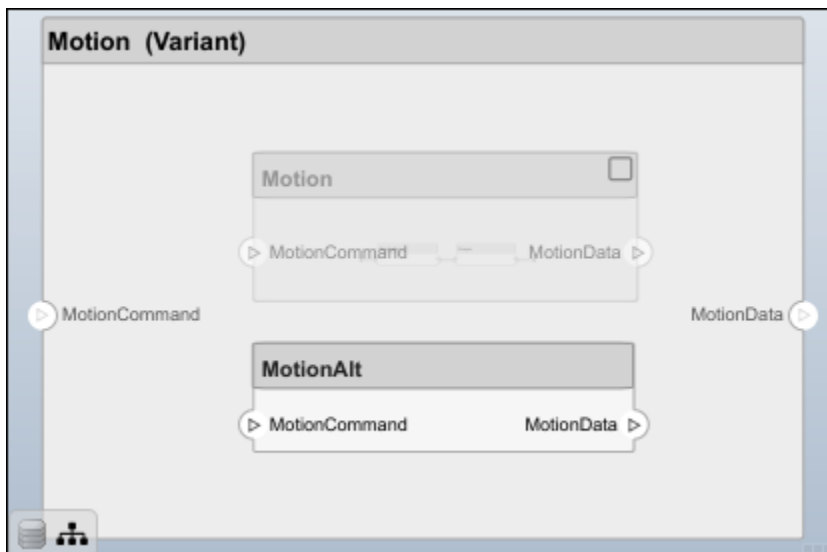
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

## Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## **Version History**

**Introduced in R2019a**

### **See Also**

editor | systemcomposer.profile.Stereotype | systemcomposer.profile.Property | loadProfile

### **Topics**

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”



# systemcomposer.profile.Property

Property in stereotype

## Description

A Property object represents properties of a stereotype in a profile for a System Composer model.

## Creation

Add a property to a stereotype using the `addProperty` function.

```
profile = systemcomposer.profile.Profile.createProfile("profileName");  
stereotype = addStereotype(profile, "stereotypeName");  
addProperty(stereotype, "propertyName", 'DefaultValue="10"')
```

## Properties

### Name — Name of property

character vector | string

Name of property, specified as a character vector or string. This property must be a valid MATLAB identifier.

Data Types: char | string

### Type — Property data type

character vector | string

Property data type, specified as a character vector or string with a valid data type.

Data Types: char | string

### Dimensions — Dimensions of property

positive integer array

Dimensions of property, specified as a positive integer array.

Data Types: double

### Min — Minimum value

numeric

Minimum value, specified as a numeric value.

Data Types: double

### Max — Maximum value

numeric

Maximum value, specified as a numeric value.

Data Types: `double`

### **Units — Property units**

character vector | string

Property units, specified as a character vector or string.

Data Types: `char` | `string`

### **Index — Property index**

numeric

Property index of the order in which the property is shown on model elements, specified as a numeric starting from one.

Data Types: `double`

### **DefaultValue — Default value of property**

string expression | array of strings

Default value of property, specified as a string expression or an array consisting of a string value and a string unit.

Data Types: `string`

### **Stereotype — Owing stereotype**

stereotype object

Owing stereotype, specified as a `systemcomposer.profile.Stereotype` object.

### **FullyQualifiedName — Qualified name of property**

character vector | string

Qualified name of property, specified as a character vector in the form '`<profile>.<stereotype>.<property>`'.

Data Types: `char`

## **Object Functions**

`destroy` Remove model element

## **Examples**

### **Build Architecture Models Programmatically**

Build an architecture model programmatically using System Composer™.

#### **Build Model**

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

## Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType", Units="dB", ...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface, "ElectricalElement", ...
    Type="electrical.electrical");
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sldd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch, "Sensor");
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, ...
    {'in', 'physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```

componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command', 'SensorPower1', 'MotionCommand'}
    {'in', 'physical', 'out'});
planningPorts(2).setInterface(physicalInterface)

componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand', 'MotionData'},...
    {'in', 'out'});

```

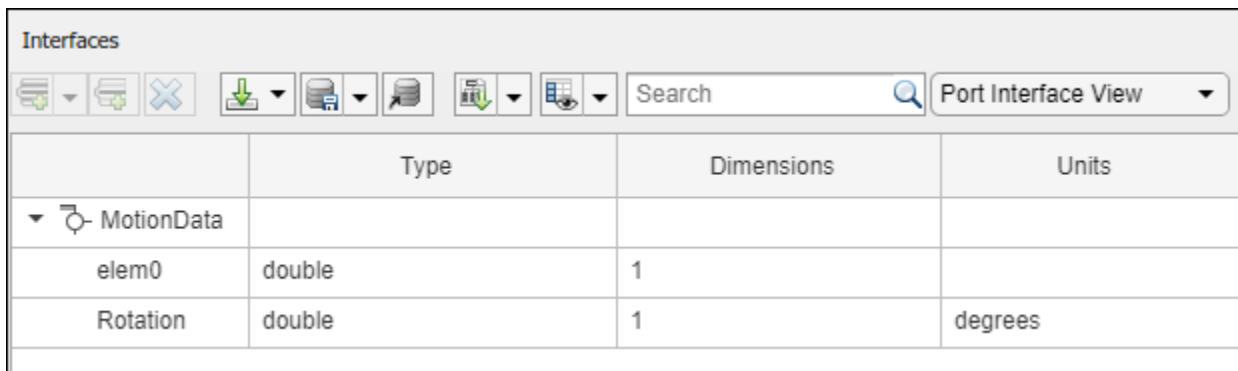
Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```

ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");

```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces			
Port Interface View			
	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```

c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);

```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```

archPort = addPort(arch, "Command", "in");

```

The connect command requires a component port as an argument. Obtain the component port, then connect.

```

compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);

```

Save the model.

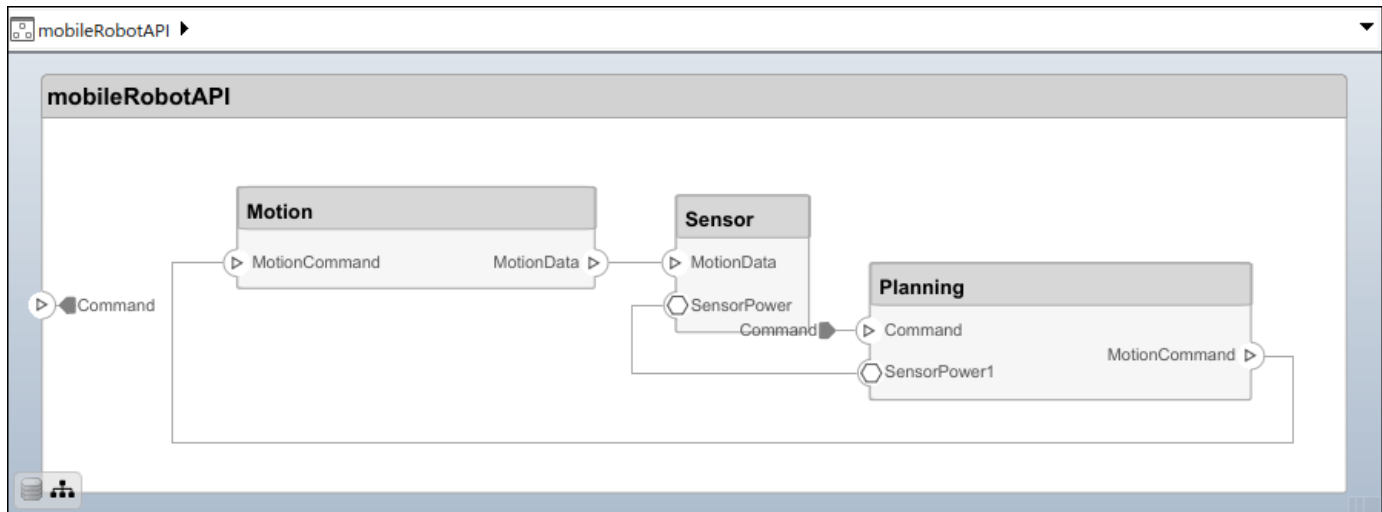
```

model.save

```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");
addProperty(elemSType, 'Description', Type="string");
addProperty(pCompSType, 'Cost', Type="double", Units="USD");
addProperty(pCompSType, 'Weight', Type="double", Units="g");
```

```
addProperty(sCompSType, 'develCost', Type="double", Units="USD");
addProperty(sCompSType, 'develTime', Type="double", Units="hour");
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

## Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture,{'controlIn','controlOut'},...
    {'in','out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');
```

```
motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');
```

```
c_planningController = connect(motionPorts(1),controllerCompPortIn);
```

For output connections, the data element must be specified.

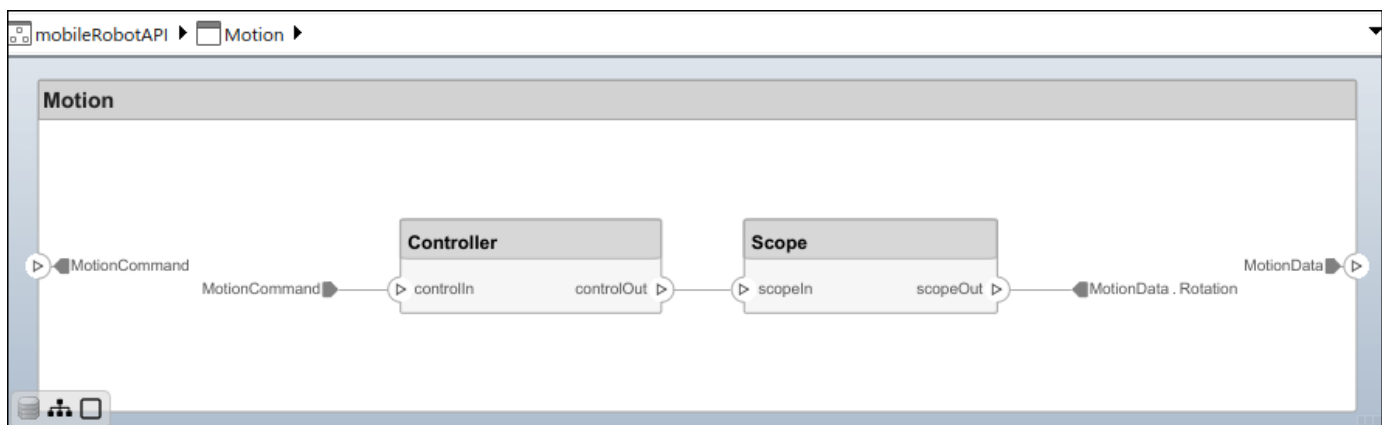
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



## Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save

linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the Planning component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

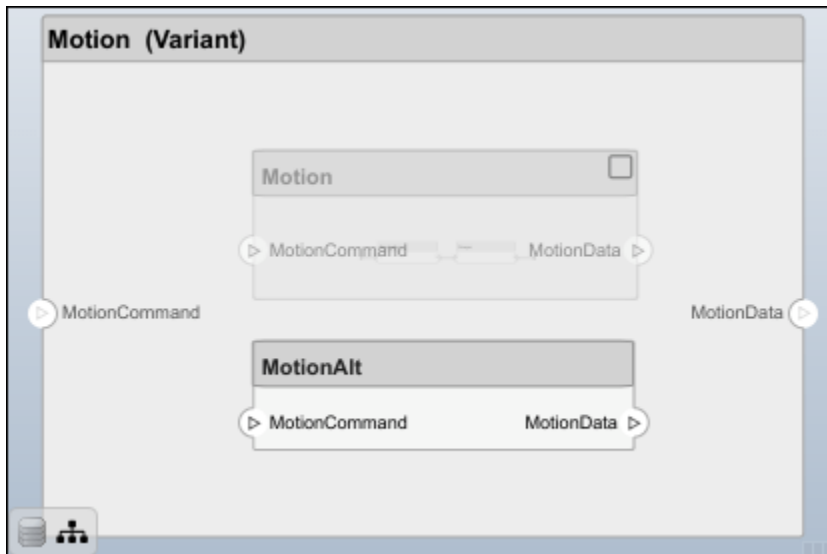
Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```





Save the model.

```
model.save
```

### **Clean Up**

Run this script to remove generated artifacts before you run this example again.

cleanUpArtifacts

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

## See Also

[systemcomposer.profile.Stereotype](#) | [systemcomposer.profile.Profile](#) | [removeProperty](#) | [addProperty](#)

## Topics

[“Define Profiles and Stereotypes”](#)

[“Use Stereotypes and Profiles”](#)

## systemcomposer.profile.Stereotype

Stereotype in profile

### Description

A Stereotype object represents stereotypes in a profile for a System Composer model.

### Creation

Add a stereotype to a profile using the `addStereotype` function.

```
profile = systemcomposer.profile.Profile.createProfile("profileName");  
addStereotype(profile,"stereotypeName");
```

### Properties

#### Name — Name of stereotype

string

Name of stereotype, specified as a string. This property must be a valid MATLAB identifier.

Example: "HardwareComponent"

Data Types: string

#### Description — Description text for stereotype

string

Description text for stereotype, specified as a string.

Data Types: string

#### Icon — Icon name for stereotype

string

Icon name for stereotype, specified as one of the following options:

- "default"
- "application"
- "channel"
- "controller"
- "database"
- "devicedriver"
- "memory"
- "network"
- "plant"

- "sensor"
- "subsystem"
- "transmitter"

This property is only valid for component stereotypes. The element a stereotype applies to is set with the `AppliesTo` property.

Data Types: `string`

### **Parent — Stereotype from which stereotype inherits properties**

`stereotype object`

Stereotype from which stereotype inherits properties, specified as a `systemcomposer.profile.Stereotype` object.

### **AppliesTo — Element type to which stereotype can be applied**

`"" (default) | "Component" | "Port" | "Connector" | "Interface" | "Function" | "Requirement" | "Link"`

Element type to which stereotype can be applied, specified as one of these options:

- `""` to apply stereotype to all element types
- "Component"
- "Port"
- "Connector"
- "Interface"
- "Function", which is only available for software architectures
- "Requirement", to be used with Requirements Toolbox
- "Link", to be used with Requirements Toolbox

Data Types: `string`

### **Abstract — Whether stereotype is abstract**

`true or 1 | false or 0`

Whether stereotype is abstract, specified as a logical. If `true`, then the stereotype cannot be directly applied on model elements, but instead serves as a parent for other stereotypes.

Data Types: `logical`

### **FullyQualifiedName — Qualified name of stereotype**

`character vector`

Qualified name of stereotype, specified as a character vector in the form `'<profile>.<stereotype>'`.

Data Types: `char`

### **ComponentHeaderColor — Component header color**

`1x3 uint32 row vector`

Component header color, specified as a `1x3 uint32` row vector in the form `[Red Green Blue]`.

This property is only valid for component stereotypes. The element a stereotype applies to is set with the `AppliesTo` property.

Example: [206 232 246]

Data Types: `uint32`

### **ConnectorLineColor — Connector line color**

1x3 `uint32` row vector

Connector line color, specified as a 1x3 `uint32` row vector in the form [Red Green Blue].

This property is only valid for connector, port, and interface stereotypes. The element a stereotype applies to is set with the `AppliesTo` property

Example: [206 232 246]

Data Types: `uint32`

### **ConnectorLineStyle — Connector line style**

character vector | string

Connector line style, specified as a character vector or string. Options include:

- "Default"
- "Dot"
- "Dash"
- "Dash Dot"
- "Dash Dot Dot"

This property is only valid for connector, port, and interface stereotypes. The element a stereotype applies to is set with the `AppliesTo` property

Data Types: `char` | `string`

### **Profile — Profile of stereotype**

profile object

Profile of stereotype from which stereotype inherits properties, specified as a `systemcomposer.profile.Profile` object.

### **Properties — Properties**

cell array of character vectors

Properties contained in stereotype and inherited from the stereotype base hierarchy, specified as a cell array of character vectors.

Data Types: `char`

### **OwnedProperties — Owned properties**

cell array of character vectors | array of strings | array of property objects

Owned properties contained in stereotype, specified as a cell array of character vectors, an array of strings, or an array of `systemcomposer.profile.Property` objects. The owned properties do not include properties inherited from the stereotype base hierarchy.

Data Types: `char` | `string`

## Object Functions

addProperty	Define custom property for stereotype
removeProperty	Remove property from stereotype
getDefaultElementStereotype	Get default stereotype for elements
setDefaultElementStereotype	Set default stereotype for elements
find	Find stereotype by name
destroy	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.slidd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType",Units="dB",...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface,"ElectricalElement",...
    Type="electrical.electrical");
linkDictionary(model,"SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sidd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch,"Sensor");
sensorPorts = addPort(componentSensor.Architecture,{'MotionData','SensorPower'},...
    {'in','physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch,"Planning");
planningPorts = addPort(componentPlanning.Architecture,{'Command','SensorPower1','MotionCommand'},...
    {'in','physical','out'});
planningPorts(2).setInterface(physicalInterface)
```

```
componentMotion = addComponent(arch,"Motion");
motionPorts = addPort(componentMotion.Architecture,{'MotionCommand','MotionData'},...
    {'in','out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.

	Type	Dimensions	Units
▼ MotionData			
elem0	double	1	
Rotation	double	1	degrees



Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch,"Command","in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

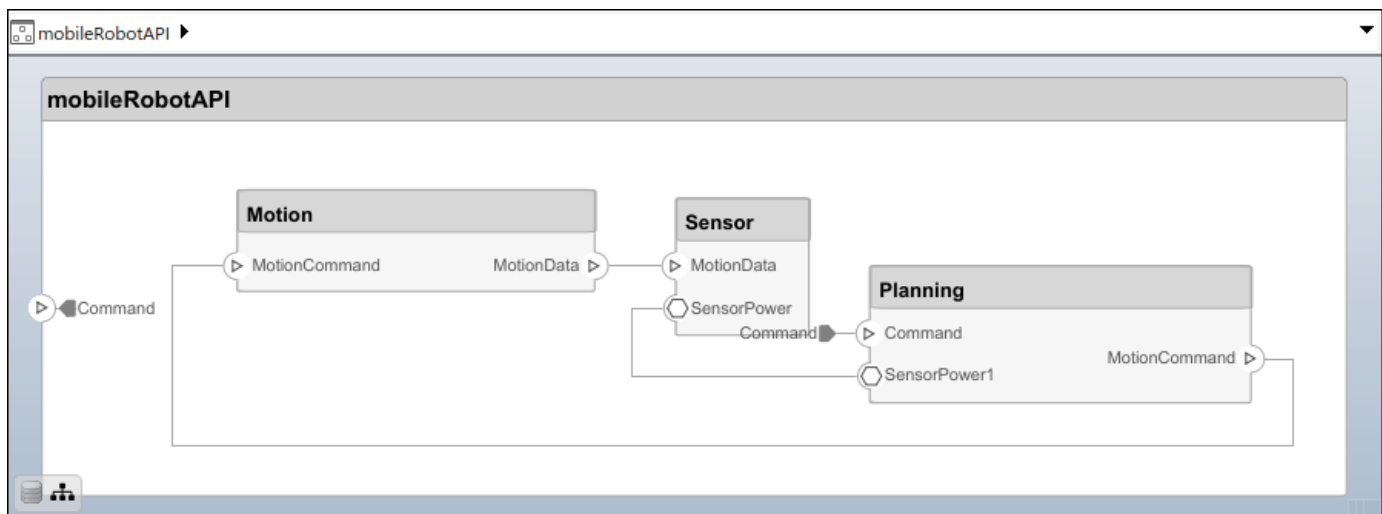
```
compPort = getPort(componentPlanning,"Command");
c_Command = connect(archPort,compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



### Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile, "projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile, "physicalComponent", AppliesTo="Component");  
sCompSType = addStereotype(profile, "softwareComponent", AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile, "standardConn", AppliesTo="Connector");
```

### **Add Properties**

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");  
addProperty(elemSType, 'Description', Type="string");  
addProperty(pCompSType, 'Cost', Type="double", Units="USD");  
addProperty(pCompSType, 'Weight', Type="double", Units="g");  
addProperty(sCompSType, 'develCost', Type="double", Units="USD");  
addProperty(sCompSType, 'develTime', Type="double", Units="hour");  
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");  
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");  
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### **Save Profile**

```
profile.save;
```

### **Apply Profile to Model**

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")  
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")  
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");  
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');  
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
```

```

    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');

```

Set the properties of connections to be identical.

```

connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end

```

### Add Hierarchy

Add two components named **Controller** and **Scope** inside the **Motion** component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```

motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);

```

For output connections, the data element must be specified.

```

c_planningScope = connect(scopeCompPortOut, motionPorts(2), DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, ...
    "GeneralProfile.standardConn");

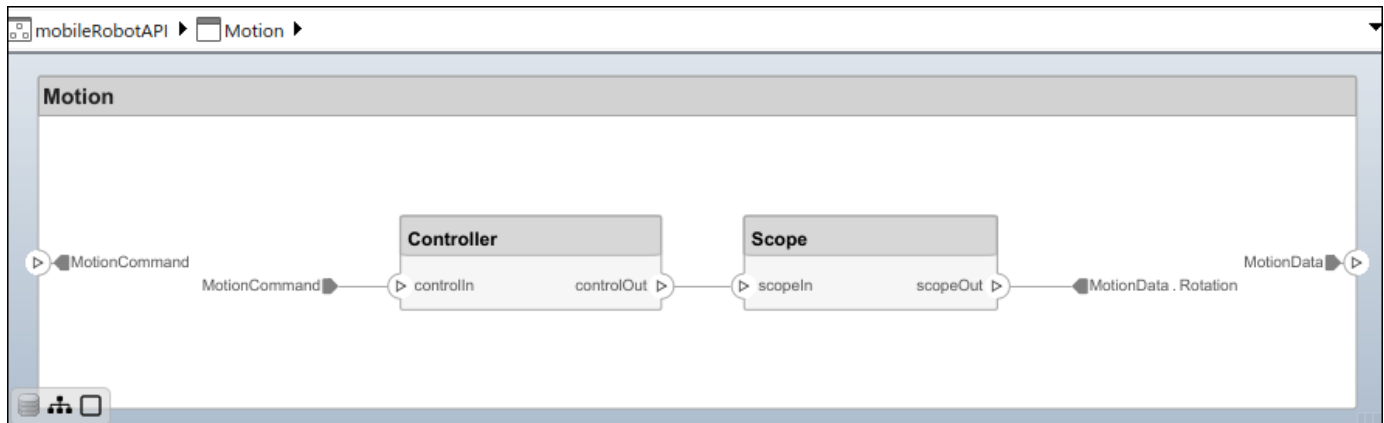
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the **Controller** component into a reference component to reference the new model. To add additional ports on the **Controller** component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "Gyroscope");
referenceModel.save
```

```
linkToModel(motionController, "mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the **Planning** component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active

choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

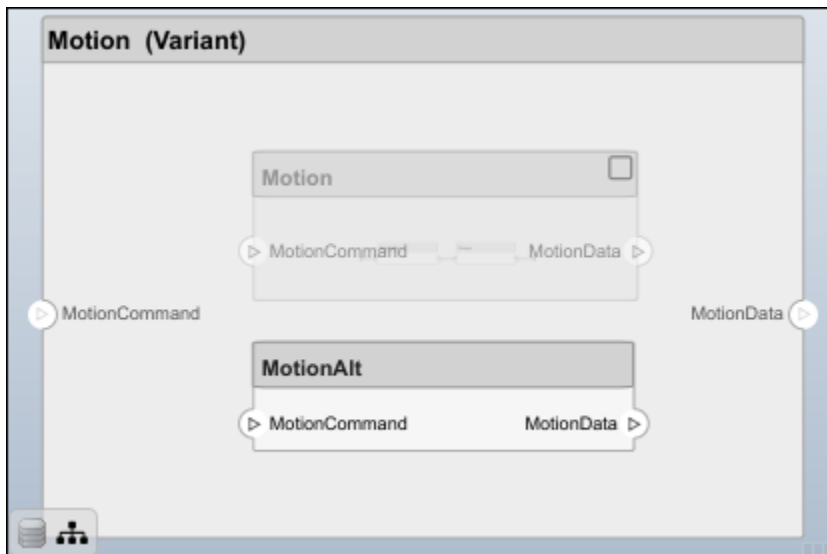
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

### Clean Up

Run this script to remove generated artifacts before you run this example again.

cleanUpArtifacts

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

## **See Also**

[addStereotype](#) | [getStereotype](#) | [removeStereotype](#) | [systemcomposer.profile.Profile](#)

## **Topics**

[“Define Profiles and Stereotypes”](#)

[“Use Stereotypes and Profiles”](#)

## systemcomposer.query.Constraint

Query constraint

### Description

The Constraint object represents all System Composer query constraints.

### Object Functions

AnyComponent	Create query to select all components in model
IsStereotypeDerivedFrom	Create query to select stereotype derived from qualified name
HasStereotype	Create query to select architectural elements with stereotype based on specified subconstraint
HasPort	Create query to select architectural elements with port based on specified subconstraint
HasConnector	Create query to select architectural elements with connector based on specified subconstraint
HasInterface	Create query to select architectural elements with interface on port based on specified subconstraint
HasInterfaceElement	Create query to select architectural elements with interface element on interface based on specified subconstraint
IsInRange	Create query to select range of property values
Property	Create query to select non-evaluated values for object properties or stereotype properties for elements
PropertyValue	Create query to select property from object or stereotype property and then evaluate property value

### Examples

#### Find Elements in Model Using Queries

Find components in a System Composer model using queries.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Open the model.

```
scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Find all the software components in the system.

```
con1 = HasStereotype(Property("Name") == "SoftwareComponent");
[compPaths, compObjs] = model.find(con1)
```

```
compPaths = 5x1 cell
    {'KeylessEntryArchitecture/Sound System/Sound Controller'
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller'}
```



```

    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
    {'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module'         }
    {'KeylessEntryArchitecture/Lighting System/Lighting Controller'           }

```

compObjs=1x5 object

1x5 Component array with properties:

```

    IsAdapterComponent
    Architecture
    ReferenceName
    Name
    Parent
    Ports
    OwnedPorts
    OwnedArchitecture
    Parameters
    Position
    Model
    SimulinkHandle
    SimulinkModelHandle
    UUID
    ExternalUID

```

Include reference models in the search.

```
softwareComps = model.find(con1, IncludeReferenceModels=true)
```

softwareComps = 9x1 cell

```

    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Sensor/Detect Door L
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Sensor/Detect Door
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Sensor/Detect Door
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Sensor/Detect Door
    {'KeylessEntryArchitecture/Sound System/Sound Controller'
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller'
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller'
    {'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module'
    {'KeylessEntryArchitecture/Lighting System/Lighting Controller'

```

Find all the base components in the system.

```
con2 = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.BaseComponent"));
```

```
baseComps = model.find(con2)
```

baseComps = 18x1 cell

```

    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Actuator' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Actuator' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Sensor'     }
    {'KeylessEntryArchitecture/FOB Locator System/Front Receiver'                     }
    {'KeylessEntryArchitecture/Engine Control System/Start//Stop Button'             }
    {'KeylessEntryArchitecture/FOB Locator System/Center Receiver'                   }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Sensor' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Sensor' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Actuator'}
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Sensor' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Actuator' }
    {'KeylessEntryArchitecture/FOB Locator System/Rear Receiver'                     }

```

```

{'KeylessEntryArchitecture/Sound System/Dashboard Speaker' }
{'KeylessEntryArchitecture/Sound System/Sound Controller' }
{'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
{'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module' }
{'KeylessEntryArchitecture/Lighting System/Lighting Controller' }

```

Find all components using the interface KeyFOBPosition.

```

con3 = HasPort(HasInterface(Property("Name") == "KeyFOBPosition"));
con3_a = HasPort(Property("InterfaceName") == "KeyFOBPosition");
keyFOBPosComps = model.find(con3)

```

```

keyFOBPosComps = 10x1 cell
{'KeylessEntryArchitecture/Sound System/Sound Controller' }
{'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
{'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module' }
{'KeylessEntryArchitecture/Lighting System/Lighting Controller' }
{'KeylessEntryArchitecture/Engine Control System' }
{'KeylessEntryArchitecture/Lighting System' }
{'KeylessEntryArchitecture/FOB Locator System' }
{'KeylessEntryArchitecture/Door Lock//Unlock System' }
{'KeylessEntryArchitecture/Sound System' }

```

Find all components whose WCET is less than or equal to 5 ms.

```

con4 = PropertyValue("AutoProfile.SoftwareComponent.WCET") <= 5;
model.find(con4)

```

```

ans = 1x1 cell array
{'KeylessEntryArchitecture/Sound System/Sound Controller'}

```

You can specify units for automatic unit conversion.

```

con5 = PropertyValue("AutoProfile.SoftwareComponent.WCET") <= Value(5,'ms');
query1Comps = model.find(con5)

```

```

query1Comps = 3x1 cell
{'KeylessEntryArchitecture/Sound System/Sound Controller' }
{'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module'}
{'KeylessEntryArchitecture/Lighting System/Lighting Controller' }

```

Find all components whose WCET is greater than 1 ms or that have a cost greater than 10 USD.

```

con6 = PropertyValue("AutoProfile.SoftwareComponent.WCET") > Value(1,'ms') | PropertyValue("AutoProfile.SoftwareComponent.Cost") > 10;
query2Comps = model.find(con6)

```

```

query2Comps = 2x1 cell
{'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }

```

Close the model.

```

model.close

```

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

find | createView | modifyQuery | runQuery | removeQuery | getQualifiedName

### Topics

“Create Architectural Views Programmatically”

# systemcomposer.ValueType

Value type in System Composer

## Description

A `ValueType` object describes a value type in System Composer. A value type can be used as a port interface or the type for a data element.

## Creation

Add a value type to a dictionary using the `addValueType` function.

```
model = systemcomposer.createModel("archModel",true);
dictionary = model.InterfaceDictionary;
airspeedType = dictionary.addValueType("AirSpeed");
```

## Properties

### Owner — Parent of value type

dictionary object | data element object | architecture port object

Parent of value type, specified as a `systemcomposer.interface.Dictionary`, `systemcomposer.interface.DataElement`, or `systemcomposer.arch.ArchitecturePort` object.

### Model — Parent model

model object

Parent System Composer model of value type, specified as a `systemcomposer.arch.Model` object.

### Name — Value type name

character vector | string

Value type name, specified as a character vector or string. This property must be a valid MATLAB identifier.

Example: "AirSpeed"

Data Types: char | string

### DataType — Value type data type

character vector | string

Value type data type, specified as a character vector or string. This property must be a valid MATLAB data type.

Data Types: char | string

### Dimensions — Value type dimensions

character vector | string

Value type dimensions, specified as a character vector or string.

Data Types: `char` | `string`

### **Units – Value type units**

character vector | string

Value type units, specified as a character vector or string.

Data Types: `char` | `string`

### **Complexity – Value type complexity**

"real" | "complex" | "auto"

Value type complexity, specified as "real", "complex", or "auto".

Data Types: `char` | `string`

### **Minimum – Value type minimum**

character vector | string

Value type minimum, specified as a character vector or string.

Data Types: `char` | `string`

### **Maximum – Value type maximum**

character vector | string

Value type maximum, specified as a character vector or string.

Data Types: `char` | `string`

### **Description – Value type description**

character vector | string

Value type description, specified as a character vector or string.

Data Types: `char` | `string`

### **UUID – Universal unique identifier**

character vector

Universal unique identifier for value type, specified as a character vector.

Example: `'91d5de2c-b14c-4c76-a5d6-5dd0037c52df'`

Data Types: `char`

### **ExternalUUID – Unique external identifier**

character vector

Unique external identifier, specified as a character vector. The external ID is preserved over the lifespan of the value type and through all operations that preserve the UUID.

Data Types: `char`

## **Object Functions**

`setName` Set name for value type, function argument, interface, or element

setDataType	Set data type for value type
setDimensions	Set dimensions for value type
setUnits	Set units for value type
setComplexity	Set complexity for value type
setMinimum	Set minimum for value type
setMaximum	Set maximum for value type
setDescription	Set description for value type or interface
applyStereotype	Apply stereotype to architecture model element
getStereotypes	Get stereotypes applied on element of architecture model
removeStereotype	Remove stereotype from model element
setProperty	Set property value corresponding to stereotype applied to element
getProperty	Get property value corresponding to stereotype applied to element
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from element
getStereotypeProperties	Get stereotype property names on element
hasStereotype	Find if element has stereotype applied
hasProperty	Find if element has property
destroy	Remove model element

## Examples

### Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

#### Build Model

To build a model, add a data dictionary with data interfaces, data elements, a value type, and a physical interface, then add components, ports, and connections. Create a profile with stereotypes and properties and then apply those stereotypes to model elements. Assign an owned interface to a port. After the model is built, you can create custom views to focus on specific considerations. You can also query the model to collect different model elements according to criteria you specify.

#### Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

```
model = systemcomposer.createModel("mobileRobotAPI");
arch = model.Architecture;
```

Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Add a physical interface and physical element with a physical domain type. Link the data dictionary to the model.

```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType", Units="dB", ...
    Description="GPS Signal Strength");
element.setType(valueType);
physicalInterface = dictionary.addPhysicalInterface("PhysicalInterface");
physicalElement = addElement(physicalInterface, "ElectricalElement", ...
```

```
Type="electrical.electrical");
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

```
systemcomposer.openModel("mobileRobotAPI");
```

View the interfaces in the Interface Editor.

	Type	Dimensions	Units	Description
▼ SensorInterfaces.sldd				
▼ GPSInterface				
SignalStrength (SignalStrengthType)	SignalStrengthType	1	dB	GPS Signal Strength
SignalStrengthType	double	1	dB	GPS Signal Strength
▼ PhysicalInterface				
ElectricalElement	Connection: foundation.electrical.electrical			

Add components, ports, and connections. Set the physical interface to the physical ports, which you will connect later.

```
componentSensor = addComponent(arch, "Sensor");
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, ...
    {'in', 'physical'});
sensorPorts(2).setInterface(physicalInterface)
```

```
componentPlanning = addComponent(arch, "Planning");
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorPower1', 'MotionCommand'}, ...
    {'in', 'physical', 'out'});
planningPorts(2).setInterface(physicalInterface)
```



```
componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, ...
    {'in', 'out'});
```

Create an owned interface on the 'MotionData' port. Add an owned data element under the owned data interface. Assign the data element "Rotation" to a value type with units set to degrees.

```
ownedInterface = motionPorts(2).createInterface("DataInterface");
ownedElement = ownedInterface.addElement("Rotation");
subInterface = ownedElement.createOwnedType(Units="degrees");
```

View the interfaces in the **Interface Editor**. Select the 'MotionData' port on the Motion component. In the Interface Editor, switch from **Dictionary View** to **Port Interface View**.



Interfaces			
 <input type="text" value="Search"/> <span>Port Interface View</span>			
	Type	Dimensions	Units
▼  MotionData			
elem0	double	1	
Rotation	double	1	degrees

Connect components with an interface rule and the default name rule. The interface rule connects ports on components that share the same interface. By default, the name rule connects ports on components that share the same name.

```
c_sensorData = connect(arch,componentSensor,componentPlanning,Rule="interface");
c_motionData = connect(arch,componentMotion,componentSensor);
c_motionCommand = connect(arch,componentPlanning,componentMotion);
```

### Add and Connect Architecture Port

Add an architecture port on the architecture.

```
archPort = addPort(arch, "Command", "in");
```

The connect command requires a component port as an argument. Obtain the component port, then connect.

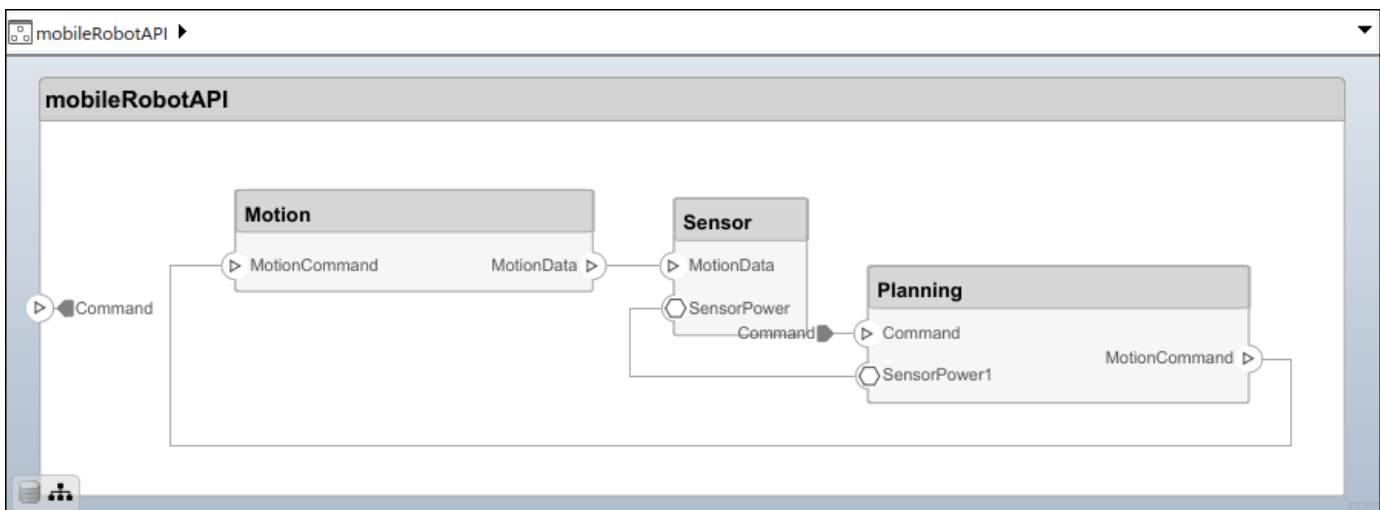
```
compPort = getPort(componentPlanning, "Command");
c_Command = connect(archPort, compPort);
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI");
```



## Create and Apply Profile with Stereotypes

Profiles are XML files that can be applied to any model. You can add stereotypes with properties to profiles and then populate the properties with specific values in the Profile Editor. Along with the built-in analysis capabilities of System Composer, stereotypes help you optimize your system for performance, cost, and reliability.

### Create Profile and Add Stereotypes

Create a profile.

```
profile = systemcomposer.createProfile("GeneralProfile");
```

Create a stereotype that applies to all element types.

```
elemSType = addStereotype(profile,"projectElement");
```

Create stereotypes for different types of components. You can select these types are based on your design needs.

```
pCompSType = addStereotype(profile,"physicalComponent",AppliesTo="Component");  
sCompSType = addStereotype(profile,"softwareComponent",AppliesTo="Component");
```

Create a stereotype for connections.

```
sConnSType = addStereotype(profile,"standardConn",AppliesTo="Connector");
```

### Add Properties

Add properties to the stereotypes. You can use properties to capture metadata for model elements and analyze nonfunctional requirements. These properties are added to all elements to which the stereotype is applied, in any model that imports the profile.

```
addProperty(elemSType, 'ID', Type="uint8");  
addProperty(elemSType, 'Description', Type="string");  
addProperty(pCompSType, 'Cost', Type="double", Units="USD");  
addProperty(pCompSType, 'Weight', Type="double", Units="g");  
addProperty(sCompSType, 'develCost', Type="double", Units="USD");  
addProperty(sCompSType, 'develTime', Type="double", Units="hour");  
addProperty(sConnSType, 'unitCost', Type="double", Units="USD");  
addProperty(sConnSType, 'unitWeight', Type="double", Units="g");  
addProperty(sConnSType, 'length', Type="double", Units="m");
```

### Save Profile

```
profile.save;
```

### Apply Profile to Model

Apply the profile to the model.

```
applyProfile(model, "GeneralProfile");
```

Apply stereotypes to components. Some components are physical components, while others are software components.

```
applyStereotype(componentPlanning, "GeneralProfile.softwareComponent")  
applyStereotype(componentSensor, "GeneralProfile.physicalComponent")  
applyStereotype(componentMotion, "GeneralProfile.physicalComponent")
```

Apply the connector stereotype to all connections.

```
batchApplyStereotype(arch, 'Connector', "GeneralProfile.standardConn");
```

Apply the general element stereotype to all connectors and ports.

```
batchApplyStereotype(arch, 'Component', "GeneralProfile.projectElement");
batchApplyStereotype(arch, 'Connector', "GeneralProfile.projectElement");
```

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', ...
    'Central unit for all sensors');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', ...
    'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', ...
    'Motor and motor controller');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

```
connections = [c_sensorData c_motionData c_motionCommand c_Command];
for k = 1:length(connections)
    setProperty(connections(k), 'GeneralProfile.standardConn.unitCost', '0.2');
    setProperty(connections(k), 'GeneralProfile.standardConn.unitWeight', '100');
    setProperty(connections(k), 'GeneralProfile.standardConn.length', '0.3');
end
```

## Add Hierarchy

Add two components named Controller and Scope inside the Motion component. Define the ports. Connect the components to the architecture and to each other, applying a connector stereotype. Hierarchy in an architecture diagram creates an additional level of detail that specifies how components behave internally.

```
motionArch = componentMotion.Architecture;

motionController = motionArch.addComponent('Controller');
controllerPorts = addPort(motionController.Architecture, {'controlIn', 'controlOut'}, ...
    {'in', 'out'});
controllerCompPortIn = motionController.getPort('controlIn');
controllerCompPortOut = motionController.getPort('controlOut');

motionScope = motionArch.addComponent('Scope');
scopePorts = addPort(motionScope.Architecture, {'scopeIn', 'scopeOut'}, {'in', 'out'});
scopeCompPortIn = motionScope.getPort('scopeIn');
scopeCompPortOut = motionScope.getPort('scopeOut');

c_planningController = connect(motionPorts(1), controllerCompPortIn);
```

For outport connections, the data element must be specified.

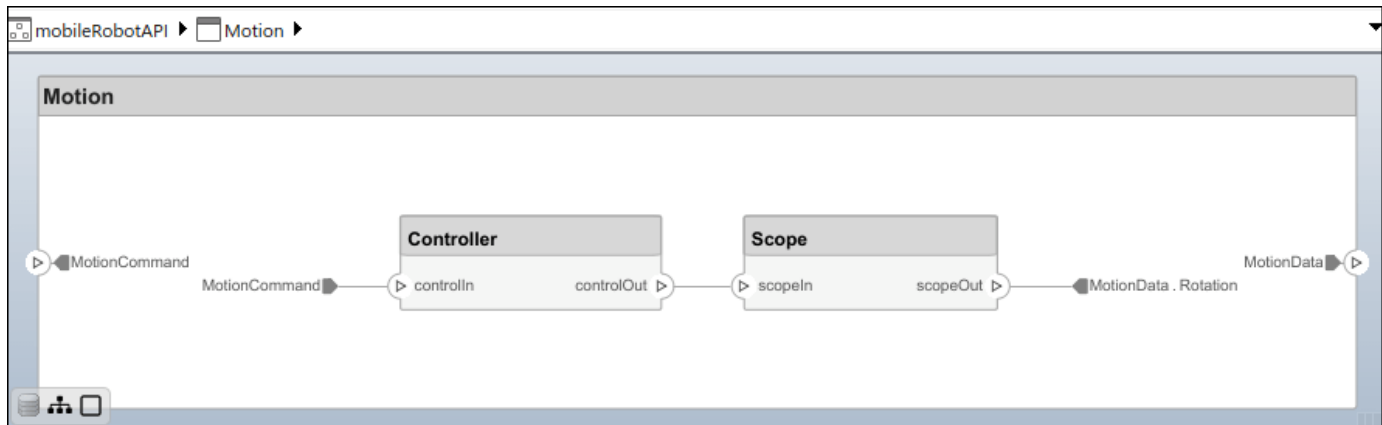
```
c_planningScope = connect(scopeCompPortOut,motionPorts(2),DestinationElement="Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,...
    "GeneralProfile.standardConn");
```

Save the model.

```
model.save
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Motion");
```



### Create Model Reference

Model references can help you organize large models hierarchically and define architectures or behaviors once that you can then reuse. When a component references another model, any existing ports on the component are removed, and ports that exist on the referenced model will appear on the component.

Create a new System Composer model. Convert the Controller component into a reference component to reference the new model. To add additional ports on the Controller component, you must update the referenced model "mobileMotion".

```
referenceModel = systemcomposer.createModel("mobileMotion");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"Gyroscope");
referenceModel.save
```

```
linkToModel(motionController,"mobileMotion");
```



Save the models.

```
referenceModel.save
model.save
```

### Make Variant Component

You can convert the `Planning` component to a variant component using the `makeVariant` function. The original component is embedded within a variant component as one of the available variant choices. You can design other variant choices within the variant component and toggle the active choice. Variant components allow you to choose behavioral designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(componentMotion);
```

Add an additional variant choice named `MotionAlt`. The second argument defines the name, and the third argument defines the label. The label identifies the choice. The active choice is controlled by the label.

```
choice2 = addChoice(variantComp,{'MotionAlt'},{'MotionAlt'});
```

Create the necessary ports on `MotionAlt`.

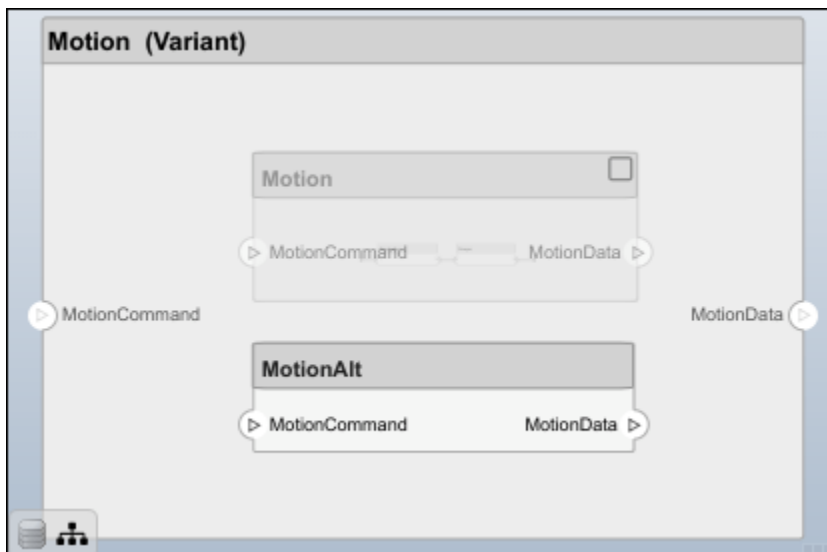
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,"MotionAlt")
```

Arrange the layout by pressing **Ctrl+Shift+A** or using this command.

```
Simulink.BlockDiagram.arrangeSystem("mobileRobotAPI/Planning");
```



Save the model.

```
model.save
```

## Clean Up

Run this script to remove generated artifacts before you run this example again.

```
cleanUpArtifacts
```

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

**See Also**

`addValueType` | `systemcomposer.interface.DataInterface` |  
`systemcomposer.interface.Dictionary` | `systemcomposer.interface.DataElement`

**Topics**

“Create Interfaces”  
“Manage Interfaces with Data Dictionaries”



# systemcomposer.view.BaseViewComponent

(Removed) View components

---

**Note** The `systemcomposer.view.BaseViewComponent` object has been removed. It has been replaced with the `systemcomposer.view.View` and the `systemcomposer.view.ElementGroup` objects. For further details, see “Compatibility Considerations”.

---

## Description

The `BaseViewComponent` object inherits from the `systemcomposer.view.ViewElement` object.

## Properties

### Name — Name of view component

character vector

Name of view component, specified as a character vector.

Example: `name = get(objBaseViewComponent, 'Name')`

Example: `set(objBaseViewComponent, 'Name', name)`

### Parent — Parent view architecture of component

view architecture object

Parent view architecture of component, specified as a `systemcomposer.view.ViewArchitecture` object.

Example: `parent = get(objBaseViewComponent, 'Parent')`

### Architecture — View architecture of component

view architecture object

View architecture of component, specified as a `systemcomposer.view.ViewArchitecture` object.

Example: `viewArch = get(objBaseViewComponent, 'ViewArchitecture')`

## Version History

**Introduced in R2019b**

### **R2021a: `systemcomposer.view.BaseViewComponent` object has been removed**

*Errors starting in R2021a*

The `systemcomposer.view.BaseViewComponent` object is removed in R2021a with the introduction of new views programmatic interfaces. For more information on how to create and edit a view using the command line, see “Create Architectural Views Programmatically”.

**See Also**

systemcomposer.view.View | createView | getView | deleteView | openViews |  
systemcomposer.view.ElementGroup

**Topics**

“Create Architecture Views Interactively”  
“Create Architectural Views Programmatically”

# systemcomposer.view.ComponentOccurrence

(Removed) Shadow of component from composition in view

---

**Note** The `systemcomposer.view.ComponentOccurrence` object has been removed. It has been replaced with the `systemcomposer.view.View` and the `systemcomposer.view.ElementGroup` objects. For further details, see “Compatibility Considerations”.

---

## Description

The `ComponentOccurrence` object inherits from the `systemcomposer.view.BaseViewComponent` object.

## Properties

### Component — Handle to composition

base component object

Handle to composition component of this occurrence, returned as a `systemcomposer.arch.BaseComponent` object.

Example: `handle = get(object, 'Component')`

## Version History

**Introduced in R2019b**

### **R2021a: systemcomposer.view.ComponentOccurrence object has been removed**

*Errors starting in R2021a*

The `systemcomposer.view.ComponentOccurrence` object is removed in R2021a with the introduction of new views programmatic interfaces. For more information on how to create and edit a view using the command line, see “Create Architectural Views Programmatically”.

## See Also

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup`

## Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# systemcomposer.view.ElementGroup

Architecture view element group

## Description

An `ElementGroup` object is used to manage element groups in architecture views for a System Composer model.

## Creation

Create a view using the `createView` function and get the `Root` property of the new `systemcomposer.view.View` object. The `Root` property returns the `systemcomposer.view.ElementGroup` that defines the view.

```
objView = createView(objModel);  
objElemGroup = objView.Root
```

## Properties

### Name — Name of element group

character vector

Name of element group, specified as a character vector.

Example: 'NewElementGroup'

Data Types: char

### UUID — Universal unique identifier

character vector

Universal unique identifier for element group, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

### Elements — Elements

array of base component objects

Elements in view, specified as a array of `systemcomposer.arch.BaseComponent` objects.

### SubGroups — Subgroups

array of element group objects

Subgroups under the parent element group, specified as an array of `systemcomposer.view.ElementGroup` objects.

## Object Functions

addElement	Add component to element group of view
removeElement	Remove component from element group of view
createSubGroup	Create subgroup in element group of view
getSubGroup	Get subgroup in element group of view
deleteSubGroup	Delete subgroup in element group of view
destroy	Remove model element

## Examples

### Create Architecture Views in System Composer with Keyless Entry System

Use a keyless entry system to programmatically create architecture views.

1. Import the package with queries.

```
import systemcomposer.query.*
```

2. Open the Simulink® project file for the Keyless Entry System.

```
scKeylessEntrySystem
```

3. Load the example model into System Composer™.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

#### Example 1: Hardware Component Review Status View

Create a filtered view that selects all hardware components in the architecture model and groups them using the ReviewStatus property.

1. Construct a query to select all hardware components.

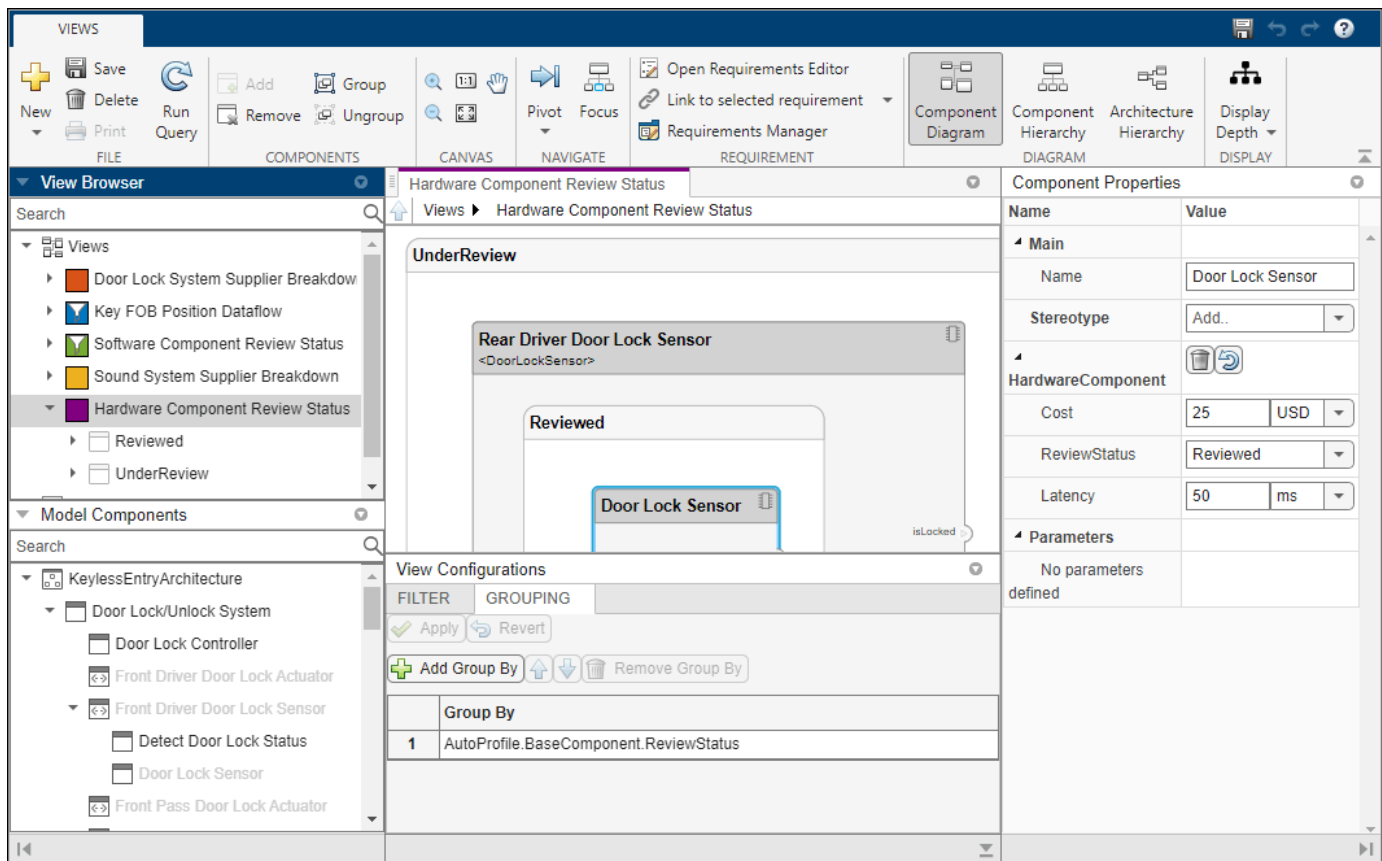
```
hwCompQuery = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent"));
```

2. Use the query to create a view.

```
model.createView("Hardware Component Review Status", ...
  Select=hwCompQuery, ...
  GroupBy={'AutoProfile.BaseComponent.ReviewStatus'}, ...
  IncludeReferenceModels=true, ...
  Color="purple");
```

3. To open the Architecture Views Gallery the **Views** section, click **Architecture Views**.

```
model.openViews
```



### Example 2: FOB Locator System Supplier View

Create a freeform view that manually pulls the components from the FOB Locator System and groups them using existing and new view components for the suppliers. In this example, you will use *element groups*, groupings of components in a view, to programmatically populate a view.

1. Create a view architecture.

```
fobSupplierView = model.createView("FOB Locator System Supplier Breakdown", ...
    Color="lightblue");
```

2. Add a subgroup called Supplier D. Add the FOB Locator Module to the view element subgroup.

```
supplierD = fobSupplierView.Root.createSubGroup("Supplier D");
supplierD.addElement("KeylessEntryArchitecture/FOB Locator System/FOB Locator Module");
```

3. Create a new subgroup for Supplier A.

```
supplierA = fobSupplierView.Root.createSubGroup("Supplier A");
```

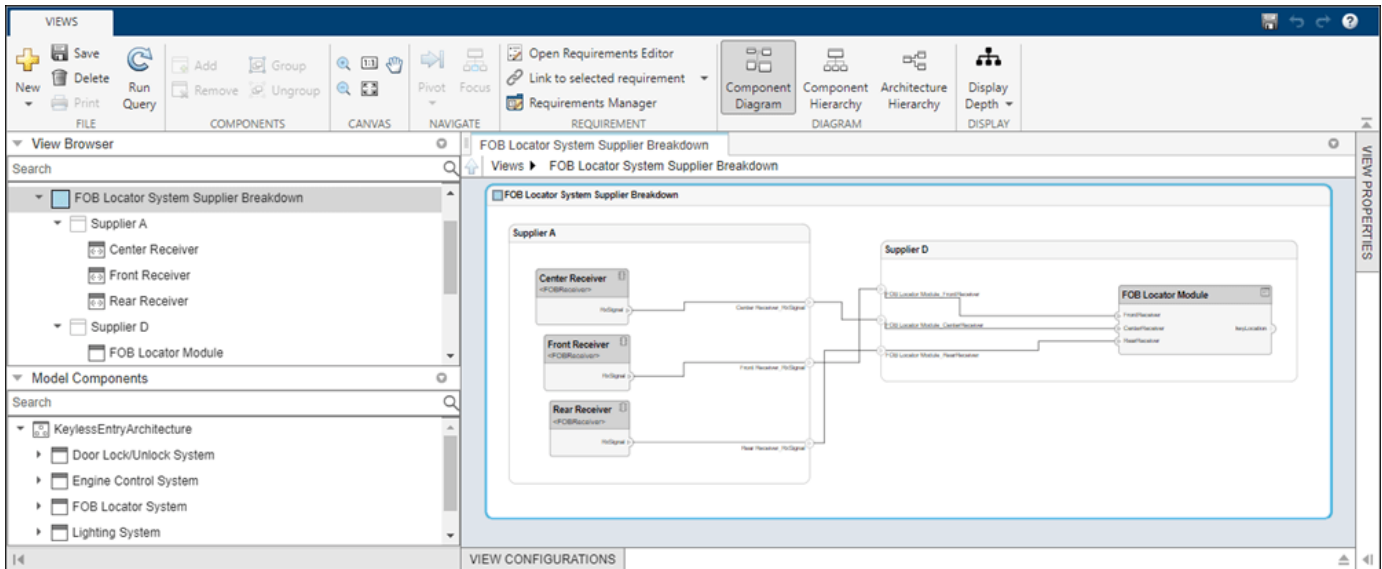
4. Add each of the FOB Receivers to view element subgroup.

```
FOBLocatorSystem = model.lookup("Path", "KeylessEntryArchitecture/FOB Locator System");
```

Find all the components which contain the name "Receiver".

```
receiverCompPaths = model.find(...
    contains(Property("Name"), "Receiver"), ...
    FOBLocatorSystem.Architecture);

supplierA.addElement(receiverCompPaths)
```



5. Save the model.

```
model.save
```

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	“Modeling System Architecture of Keyless Entry System”

Term	Definition	Application	More Information
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

openViews | createView | getView | deleteView | systemcomposer.view.View | getQualifiedName

### Topics

“Create Architecture Views Interactively”  
“Create Architectural Views Programmatically”



“Display Component Hierarchy and Architecture Hierarchy Using Views”

## **systemcomposer.view.View**

Architecture view

### **Description**

A View object is used to manage architecture views for a System Composer model.

### **Creation**

Create a view using the `createView` function.

```
objView = createView(objModel)
```

### **Properties**

#### **Name — Name of view**

string

Name of view, specified as a string.

Example: "NewView"

Data Types: string

#### **Root — Root element group**

element group object

Root element group that defines view, specified as a `systemcomposer.view.ElementGroup` object.

#### **Model — Architecture model**

model object

Architecture model where view belongs, specified as a `systemcomposer.arch.Model` object.

#### **UUID — Universal unique identifier**

character vector

Universal unique identifier for view, specified as a character vector.

Example: '91d5de2c-b14c-4c76-a5d6-5dd0037c52df'

Data Types: char

#### **Select — Selection query**

constraint object

Selection query associated with view, specified as a `systemcomposer.query.Constraint` object.

#### **GroupBy — Grouping criteria**

string array of properties

Grouping criteria, specified as a string array of properties in the form "`<profile>.<stereotype>.<property>`".

Example:

```
["AutoProfile.MechanicalComponent.mass", "AutoProfile.MechanicalComponent.cost"]
```

### Color — Color of view

string

Color of view, specified as a string. The color can be the name "blue", "black", or "green", or it can be an RGB value encoded in a hexadecimal string: "#FF00FF" or "#DDDDDD". An invalid color results in an error.

### Description — Description of view

string

Description of view, specified as a string.

Data Types: `string`

### IncludeReferenceModels — Whether to include referenced models

true or 1 | false or 0

Whether to include referenced models, specified as a logical.

Example: `included = get(objView, 'IncludeReferenceModels')`

Data Types: `logical`

## Object Functions

<code>modifyQuery</code>	Modify architecture view query and property groupings
<code>runQuery</code>	Re-run architecture view query on model
<code>removeQuery</code>	Remove architecture view query
<code>destroy</code>	Remove model element

## Examples

### Create Architecture Views in System Composer with Keyless Entry System

Use a keyless entry system to programmatically create architecture views.

1. Import the package with queries.

```
import systemcomposer.query.*
```

2. Open the Simulink® project file for the Keyless Entry System.

```
scKeylessEntrySystem
```

3. Load the example model into System Composer™.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

### Example 1: Hardware Component Review Status View

Create a filtered view that selects all hardware components in the architecture model and groups them using the ReviewStatus property.

1. Construct a query to select all hardware components.

```
hwCompQuery = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent"));
```

2. Use the query to create a view.

```
model.createView("Hardware Component Review Status",...
  Select=hwCompQuery,...
  GroupBy={'AutoProfile.BaseComponent.ReviewStatus'},...
  IncludeReferenceModels=true,...
  Color="purple");
```

3. To open the Architecture Views Gallery the **Views** section, click **Architecture Views**.

```
model.openViews
```

The screenshot shows the software interface with the following components:

- Views Browser:** Shows a tree of views, with 'Hardware Component Review Status' selected and expanded to show 'Reviewed' and 'UnderReview' sub-views.
- Model Components:** Shows a tree of model components, including 'Door Lock/Unlock System', 'Door Lock Controller', 'Front Driver Door Lock Actuator', 'Front Driver Door Lock Sensor', 'Detect Door Lock Status', 'Door Lock Sensor', and 'Front Pass Door Lock Actuator'.
- Hardware Component Review Status View:** Displays a diagram with a container 'UnderReview' containing a container 'Rear Driver Door Lock Sensor' (stereotyped as <DoorLockSensor>). Inside 'Rear Driver Door Lock Sensor' is a container 'Reviewed' containing a component 'Door Lock Sensor'.
- Component Properties:** Shows properties for the selected 'Door Lock Sensor':
 

Name	Value
Name	Door Lock Sensor
Stereotype	Add..
HardwareComponent	
Cost	25 USD
ReviewStatus	Reviewed
Latency	50 ms
Parameters	No parameters defined
- View Configurations:** Shows the view is filtered and grouped by 'AutoProfile.BaseComponent.ReviewStatus'.

### Example 2: FOB Locator System Supplier View

Create a freeform view that manually pulls the components from the FOB Locator System and groups them using existing and new view components for the suppliers. In this example, you will use *element groups*, groupings of components in a view, to programmatically populate a view.

1. Create a view architecture.

```
fobSupplierView = model.createView("FOB Locator System Supplier Breakdown",...
    Color="lightblue");
```

2. Add a subgroup called Supplier D. Add the FOB Locator Module to the view element subgroup.

```
supplierD = fobSupplierView.Root.createSubGroup("Supplier D");
supplierD.addElement("KeylessEntryArchitecture/FOB Locator System/FOB Locator Module");
```

3. Create a new subgroup for Supplier A.

```
supplierA = fobSupplierView.Root.createSubGroup("Supplier A");
```

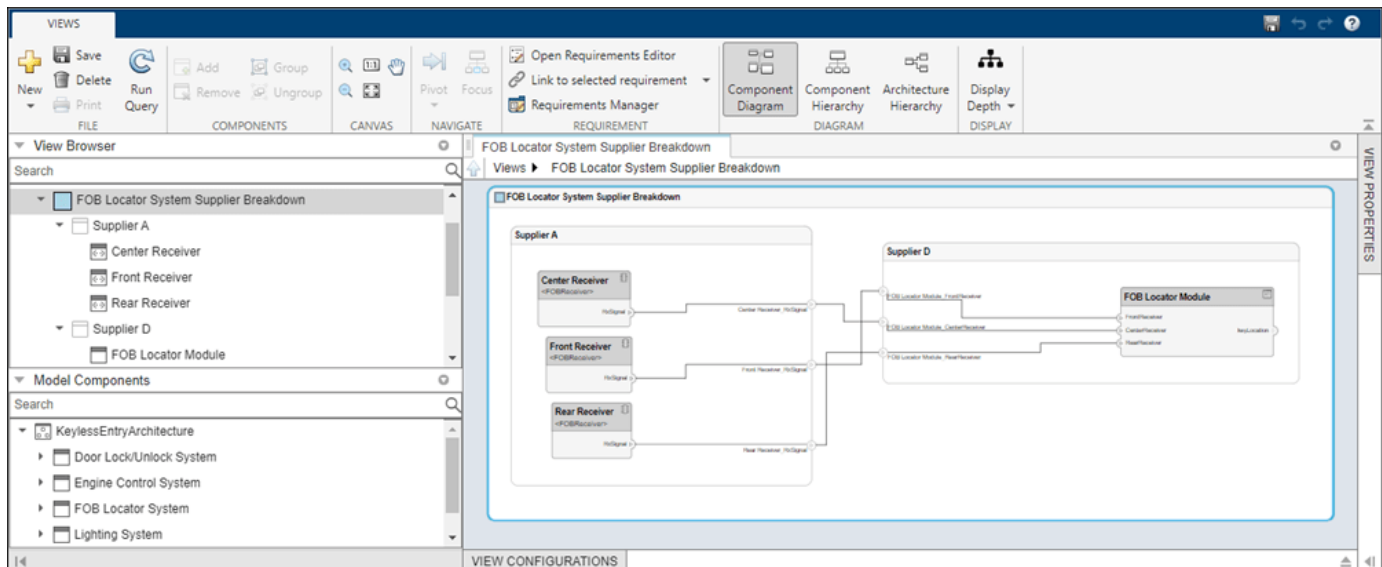
4. Add each of the FOB Receivers to view element subgroup.

```
FOBLocatorSystem = model.lookup("Path", "KeylessEntryArchitecture/FOB Locator System");
```

Find all the components which contain the name "Receiver".

```
receiverCompPaths = model.find(...
    contains(Property("Name"), "Receiver"),...
    FOBLocatorSystem.Architecture);
```

```
supplierA.addElement(receiverCompPaths)
```



5. Save the model.

model.save

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

`openViews` | `createView` | `getView` | `deleteView` | `systemcomposer.view.ElementGroup` | `getQualifiedName`

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

“Display Component Hierarchy and Architecture Hierarchy Using Views”

## systemcomposer.view.ViewArchitecture

(Removed) Set of view components in architecture view

---

**Note** The `systemcomposer.view.ViewArchitecture` class has been removed. It has been replaced with the `systemcomposer.view.View` and the `systemcomposer.view.ElementGroup` classes. For further details, see “Compatibility Considerations”.

---

### Description

A `ViewArchitecture` object describes a set of view components that make up a view. This object inherits from the `systemcomposer.view.ViewElement` object.

### Properties

#### Name — Name of architecture

character vector

Name of architecture derived from the parent component or model name to which the architecture belongs, returned as a character vector.

Example: `name = get(objViewArchitecture, 'Name')`

Data Types: `char`

#### IncludeReferenceModels — Control inclusion of referenced models

`true` or `1` | `false` or `0`

Control inclusion of referenced models, returned as a logical with values `1` (`true`) or `0` (`false`).

Example: `included = get(objViewArchitecture, 'IncludeReferenceModels')`

Data Types: `logical`

#### Color — Color of view architecture

character vector

Color of view architecture, returned as a character vector as a name `'blue'`, `'black'`, or `'green'` or as a RGB value encoded in a hexadecimal string `'#FF00FF'` or `'#DDDDDD'`. An invalid color string results in an error.

Example: `color = get(objViewArchitecture, 'Color')`

#### Description — Description of view architecture

character vector

Description of view architecture, returned as a character vector.

Example: `description = get(objViewArchitecture, 'Description')`

Example: `set(objViewArchitecture, 'Description', description)`

Data Types: `char`



**Parent — Component that owns view architecture**

base view component object

Component that owns view architecture, returned as a `systemcomposer.view.BaseViewComponent` object. For a root view architecture, returns an empty handle.

Example: `parentComponent = get(objViewArchitecture, 'Parent')`

**Components — Array of handles to child components**

array of base view component objects

Array of handles to the set of child components of this view architecture, returned as an array of `systemcomposer.view.BaseViewComponent` objects.

Example: `childComponents = get(objViewArchitecture, 'Components')`

**Methods**

`addComponent` (Removed) Add component to view given path  
`removeComponent` (Removed) Remove component from view  
`createViewComponent` (Removed) Create view component

**Version History****Introduced in R2019b****R2021a: systemcomposer.view.ViewArchitecture object has been removed***Errors starting in R2021a*

The `systemcomposer.view.ViewArchitecture` object is removed in R2021a with the introduction of new views programmatic interfaces. For more information on how to create and edit a view using the command line, see “Create Architectural Views Programmatically”.

**See Also**

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup`

**Topics**

“Create Architecture Views Interactively”  
 “Create Architectural Views Programmatically”

# systemcomposer.view.ViewComponent

(Removed) View component within architecture view

---

**Note** The `systemcomposer.view.ViewComponent` class has been removed. It has been replaced with the `systemcomposer.view.View` and the `systemcomposer.view.ElementGroup` classes. For further details, see “Compatibility Considerations”.

---

## Description

A `ViewComponent` object is a component that exists only in the view in which it is created. These components do not exist in the composition. This object inherits from the `systemcomposer.view.BaseViewComponent` object.

## Version History

**Introduced in R2019b**

**R2021a: `systemcomposer.view.ViewComponent` object has been removed**

*Errors starting in R2021a*

The `systemcomposer.view.ViewComponent` object is removed in R2021a with the introduction of new views programmatic interfaces. For more information on how to create and edit a view using the command line, see “Create Architectural Views Programmatically”.

## See Also

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup`

## Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# systemcomposer.view.ViewElement

(Removed) All view elements

---

**Note** The `systemcomposer.view.ViewElement` object has been removed. It has been replaced with the `systemcomposer.view.View` and the `systemcomposer.view.ElementGroup` objects. For further details, see “Compatibility Considerations”.

---

## Description

Base class of all view elements.

## Properties

### ZCIdentifier — Identifier of object

character vector

Identifier of object. This property is used by Simulink Requirements™.

Example: `identifier = get(objViewElement, 'ZCIdentifier')`

Data Types: `char`

## Version History

**Introduced in R2009b**

### **R2021a: systemcomposer.view.ViewElement object has been removed**

*Errors starting in R2021a*

The `systemcomposer.view.ViewElement` object is removed in R2021a with the introduction of new views programmatic interfaces. For more information on how to create and edit a view using the command line, see “Create Architectural Views Programmatically”.

## See Also

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup`

## Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”



# Classes

---

## systemcomposer.rptgen.finder.AllocationListFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find allocations

### Description

The `systemcomposer.rptgen.finder.AllocationListFinder` class searches for all the components to and from which a particular component has been allocated in a System Composer architecture model.

### Creation

`finder = AllocationListFinder(Container)` creates a finder that finds all allocations to and from the component defined by the `ComponentName` property.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

### Properties

#### Container — Allocation set file

string

Allocation set file with the `.mldatx` extension, specified as a string.

Example: `f = AllocationListFinder("AllocationSet.mldatx")`

Data Types: string

#### ComponentName — Component to and from which to find allocations

string

Component to and from which to find allocations, specified as a string of the full path.

Example: `f.ComponentName = "mTestModel/Component1"`

#### Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

Data Types: string

### Properties — Properties of objects to find

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: char

## Methods

### Public Methods

`find` Find allocations to and from component  
`next` Get next allocation list search result  
`hasNext` Determine if allocation list search result queue is nonempty

## Examples

### Generate AllocationList Finder Report

Use the `AllocationListFinder` and `AllocationListResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationListFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocations"));
add(rpt,TableOfContents);

allocationListFinder = AllocationListFinder("AllocationSet.mldatx");
allocationListFinder.ComponentName = "mTestModel/Component1";
chapter = Chapter("Title","Allocations");
while hasNext(allocationListFinder)
    allocations = next(allocationListFinder);
    sect = Section("Title",allocationListFinder.ComponentName);
    add(sect,allocations);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

**See Also**

systemcomposer.rptgen.finder.AllocationListResult |  
systemcomposer.rptgen.report.AllocationList | find | next | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”



# systemcomposer.rptgen.finder.AllocationListResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for allocations

## Description

Allocation list search result object for a component in a System Composer architecture model.

The systemcomposer.rptgen.finder.AllocationListResult class is a handle class.

## Creation

`result = AllocationListResult` creates a search result object for allocations to and from a specific component found by a systemcomposer.rptgen.finder.AllocationListFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.AllocationListFinder class creates objects of this type for each allocation that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### AllocatedFrom — Components from which specified component has been allocated

array of strings

Components from which specified component has been allocated, returned as an array of strings.

Data Types: string

### AllocatedTo — Components to which specified component has been allocated

array of strings

Components to which specified component has been allocated, returned as an array of strings.

Data Types: string

### Tag — Tag to associate with result

string

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

## Methods

### Public Methods

`getReporter` Get allocation list reporter

## Examples

### Generate AllocationList Result Report

Use the `AllocationListFinder` and `AllocationListResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationListResultReport", ...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocations"));
add(rpt,TableOfContents);

allocationListFinder = AllocationListFinder("AllocationSet.mldatx");
allocationListFinder.ComponentName = "mTestModel/Component1";
chapter = Chapter("Title",allocationListFinder.ComponentName);
result = find(allocationListFinder);
reporter = getReporter(result);

add(rpt,chapter);
append(rpt,reporter);
close(rpt);
rptview(rpt)
```

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.AllocationListFinder` |  
`systemcomposer.rptgen.report.AllocationList` | `find` | `next` | `hasNext` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.AllocationSetFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find allocation sets

## Description

The `systemcomposer.rptgen.finder.AllocationSetFinder` class searches for information about a given allocation set in a System Composer architecture model.

## Creation

`finder = AllocationSetFinder(Container)` creates a finder that finds information about an allocation set.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

## Properties

### Container — Allocation set file

string

Allocation set file with the `.mldatx` extension, specified as a string.

Example: `f = AllocationSetFinder("AllocationSet.mldatx")`

Data Types: string

### Properties — Properties of objects to find

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain', '5'}`

Data Types: char

## Methods

### Public Methods

**find** Find information about allocation set  
**hasNext** Determine if allocation set search result queue is nonempty  
**next** Get next allocation set search result

## Examples

### Generate AllocationSet Finder Report

Use the AllocationSetFinder and AllocationSetResult classes to generate a report.

```

import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationSetFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocation Sets"));
add(rpt,TableOfContents);

allocationSetFinder = AllocationSetFinder("AllocationSet.mldatx");
chapter = Chapter("Title","Allocation Set");

while hasNext(allocationSetFinder)
  allocationSets = next(allocationSetFinder);
  sect = Section(strcat("Allocations in ",allocationSets.Name));
  add(sect,allocationSets);
  add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
  
```

## Version History

**Introduced in R2022b**

### See Also

[systemcomposer.rptgen.finder.AllocationSetResult](#) |  
[systemcomposer.rptgen.report.AllocationSet](#) | [find](#) | [hasNext](#) | [next](#) | [getReporter](#) |  
[createTemplate](#) | [customizeReporter](#) | [getClassFolder](#)

### Topics

“System Composer Report Generation for System Architectures”  
 “System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.AllocationSetResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for allocation sets

## Description

Allocation set search result object in a System Composer architecture model.

The systemcomposer.rptgen.finder.AllocationSetResult class is a handle class.

## Creation

`result = AllocationSetResult` creates a search result object for an allocation set found by a systemcomposer.rptgen.finder.AllocationSetFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.AllocationSetFinder class creates objects of this type for each allocation set that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### Name — Name of allocation set

string

Name of allocation set, returned as a string.

Data Types: string

### SourceModel — Source model of allocation set

string

Source model of allocation set, returned as a string.

Data Types: string

### TargetModel — Target model of allocation set

string

Target model of allocation set, returned as a string.

Data Types: `string`

### **Description — Description of allocation set**

`string`

Description of allocation set, returned as a string.

Data Types: `string`

### **Scenarios — Scenarios present in allocation set**

structure with fields

Scenarios present in allocation set, returned as a structure with fields:

- `Name`, returned as a string.
- `Allocations`, returned as a structure with fields:
  - `SourceElement`, returned as the fully qualified name of the source component allocated from.
  - `TargetElement`, returned as the fully qualified name of the target component allocated to.
- `UUID`, or universal unique identifier of the scenario, returned as a string.

Data Types: `struct`

### **Tag — Tag to associate with result**

`string`

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

## **Methods**

### **Public Methods**

`getReporter` Get allocation set reporter

Use the `AllocationSetFinder` and `AllocationSetResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationSetResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocation Sets"));
add(rpt,TableOfContents);
chapter = Chapter("Title","Allocation Sets");

allocationSetFinder = AllocationSetFinder("AllocationSet.mldatx");
result = find(allocationSetFinder);
reporter = getReporter(result);

add(rpt,chapter);
```

```
append(rpt, reporter);  
close(rpt);  
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationSetFinder |  
systemcomposer.rptgen.report.AllocationSet | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## systemcomposer.rptgen.finder.ComponentFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find components

### Description

The `systemcomposer.rptgen.finder.ComponentFinder` class searches for information about all the components in a System Composer architecture model.

### Creation

`finder = ComponentFinder(Container)` creates a finder that finds all components in a model that meet the `Query` property.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

### Properties

#### Container – Architecture model file name

string

Architecture model file name without the `.slx` extension, specified as a string.

Example: `f = ComponentFinder("ArchModel")`

Data Types: `string`

#### Query – Query to find components

constraint object

Query to find components, specified as a `systemcomposer.query.Constraint` object.

#### Attributes:

GetAccess	public
SetAccess	public

#### Recurse – Option to recursively search model

true or 1 (default) | false or 0



Option to recursively search model or to only search a specific layer, specified as 1 (`true`) to recursively search or 0 (`false`) to only search the specific layer.

**Attributes:**

```
GetAccess          public
SetAccess          public
```

Data Types: `logical`

**IncludeReferenceModels — Option to search for reference architectures**

`false` or 0 (default) | `true` or 1

Option to search for reference architectures, specified as a logical.

**Attributes:**

```
GetAccess          public
SetAccess          public
```

Data Types: `logical`

**Properties — Properties of objects to find**

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: `char`

## Methods

**Public Methods**

```
find      Find information about component
hasNext   Determine if component search result queue is nonempty
next      Get next component search result
```

## Examples

**Generate Component Finder Report**

Use the `ComponentFinder` and `ComponentResult` classes to generate a report.

```
import systemcomposer.rptgen.finder.*
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.query.*

rpt = slreportgen.report.Report(output="ComponentFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Components"));
add(rpt,TableOfContents);
```

```
componentFinder = ComponentFinder("mTestModel");
componentFinder.Query = AnyComponent;

chapter = Chapter("Components in mTestModel");

while hasNext(componentFinder)
    componentResult = next(componentFinder);
    sect = Section(componentResult.Name);
    add(sect, componentResult);
    add(chapter, sect);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ComponentResult |  
systemcomposer.rptgen.report.Component | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.ComponentResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for components

## Description

Search result object for information about a component in a System Composer architecture model.

The systemcomposer.rptgen.finder.ComponentResult class is a handle class.

## Creation

`result = ComponentResult` creates a search result object for a component found by a systemcomposer.rptgen.finder.ComponentFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.ComponentFinder class creates objects of this type for each component that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### Name — Name of component

string

Name of component, returned as a string.

Data Types: string

### Parent — Parent of component

string

Parent of component, returned as a string.

Data Types: string

### Children — Children of component

array of component result objects

Children of component, returned as an array of `systemcomposer.rptgen.finder.ComponentResult` objects.

**Ports — Ports on component**

array of component port objects

Ports on component, returned as an array of `systemcomposer.arch.ComponentPort` objects.

**ReferenceName — Reference model name used by component**

string

Reference model name used by component, returned as a string.

Data Types: `string`

**Kind — Kind of AUTOSAR component**

string

Kind of AUTOSAR component, returned as a string.

Data Types: `string`

**Tag — Tag to associate with result**

string

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

**Methods****Public Methods**

`getReporter` Get component reporter

**Examples****Generate Component Result Report**

Use the `ComponentFinder` and `ComponentResult` classes to generate a report.

```
import systemcomposer.rptgen.finder.*
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.query.*

rpt = slreportgen.report.Report(output="ComponentResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Components"));
add(rpt,TableOfContents);
chapter = Chapter("Title","Components");

componentFinder = ComponentFinder("mTestModel");
componentFinder.Query = AnyComponent;
result = find(componentFinder);
```

```
for i = result
    reporter = getReporter(i);
    reporter.IncludeProperties = false;
    reporter.IncludeSnapshot = false;
    add(chapter, reporter);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.finder.ComponentFinder` |  
`systemcomposer.rptgen.report.Component` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## systemcomposer.rptgen.finder.ConnectorFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find connectors

### Description

The `systemcomposer.rptgen.finder.ConnectorFinder` class searches for information about all the connectors in a given System Composer architecture model.

### Creation

`finder = ConnectorFinder(Container)` creates a finder that finds all connectors in a component or on an architecture specified by the `Filter` property. The component is defined by the `ComponentName` property.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

### Properties

#### Container – Architecture model file name

string

Architecture model file name without the `.slx` extension, specified as a string.

Example: `f = ConnectorFinder("ArchModel")`

Data Types: `string`

#### Filter – Filter to find connectors

"Architecture" (default) | "Component"

Filter to find connectors, specified as "Component" to find connectors in a component or "Architecture" to find connectors on an architecture.

#### Attributes:

GetAccess	public
SetAccess	public

Data Types: string

### ComponentName — Component to find connectors in

string

Component to find connectors in, specified as a string of the full path.

Example: `f.ComponentName = "mTestModel/Component1"`

#### Attributes:

GetAccess	public
SetAccess	public

Data Types: string

### Properties — Properties of objects to find

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: char

## Methods

### Public Methods

find	Find information about connector
hasNext	Determine if connector search result queue is nonempty
next	Get next connector search result

## Examples

### Generate Connector Finder Report

Use the ConnectorFinder and ConnectorResult classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ConnectorFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Connectors in %s Model',model_name)));
add(rpt,TableOfContents);

connectorFinder = ConnectorFinder(model_name);
connectorFinder.ComponentName = "scExampleSmallUAVModel/Flight Support Components/GPS Module";
connectorFinder.Filter = "Component";
chapter = Chapter("Title","Connectors");
```

```
while hasNext(connectorFinder)
  connector = next(connectorFinder);
  sect = Section("Title",connector.Name);
  add(sect,connector);
  add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.ConnectorResult |  
systemcomposer.rptgen.report.Connector | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”



# systemcomposer.rptgen.finder.ConnectorResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for connectors

## Description

Search result object for information about a connector in a System Composer architecture model.

The systemcomposer.rptgen.finder.ConnectorResult class is a handle class.

## Creation

result = ConnectorResult creates a search result object for a connector found by a systemcomposer.rptgen.finder.ConnectorFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.ConnectorFinder class creates objects of this type for each connector that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### Name — Name of connector

string

Name of connector, returned as a string.

Data Types: string

### Parent — Parent component of connector

string

Parent component of connector, returned as a string.

Data Types: string

### SourcePort — Source port of connector

string

Source port of connector, returned as a string.

Data Types: string

### **DestinationPort — Destination port of connector**

string

Destination port of connector, returned as a string.

Data Types: string

### **Stereotypes — Stereotypes on connector**

array of strings

Stereotypes on connector, returned as an array of strings.

Data Types: string

### **Tag — Tag to associate with result**

string

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: string

## **Methods**

### **Public Methods**

getReporter Get connector reporter

## **Examples**

### **Generate Connector Result Report**

Use the ConnectorFinder and ConnectorResult classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ConnectorResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Connectors in %s Model',model_name)));
add(rpt,TableOfContents);

connectorFinder = ConnectorFinder(model_name);
connectorFinder.Filter = "Component";
connectorFinder.ComponentName = "scExampleSmallUAVModel/Flight Support Components";
chapter = Chapter("Title","Connectors");
result = find(connectorFinder);
add(rpt,chapter);

for r = result
```

```
    reporter = getReporter(r);  
    append(rpt, reporter);  
end
```

```
close(rpt);  
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ConnectorFinder |  
systemcomposer.rptgen.report.Connector | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## systemcomposer.rptgen.finder.DictionaryFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find dictionaries

### Description

The `systemcomposer.rptgen.finder.DictionaryFinder` class searches for information about all the dictionaries in a given System Composer architecture model.

### Creation

`finder = DictionaryFinder(Container)` creates a finder that finds all dictionaries in an architecture model specified by the `Type` property to search for model dictionaries or reference dictionaries.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

### Properties

#### Container – Architecture model file name

string

Architecture model file name without the `.slx` extension, specified as a string.

Example: `f = DictionaryFinder("ArchModel")`

Data Types: `string`

#### Type – Filter to find dictionaries

"Model" | "Dictionary"

Filter to find dictionaries, specified as "Model" to find dictionaries in the model or "Dictionary" to find reference dictionaries.

#### Attributes:

GetAccess	public
SetAccess	public

Data Types: string

### Properties — Properties of objects to find

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: char

## Methods

### Public Methods

`find` Find information about dictionary  
`hasNext` Determine if dictionary search result queue is nonempty  
`next` Get next dictionary search result

## Examples

### Generate Dictionary Finder Report

Use the `DictionaryFinder` and `DictionaryResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

sckeylessEntrySystem
model_name = "KeylessEntryArchitecture";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="DictionaryFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Dictionaries in %s Model',model_name)));
add(rpt,TableOfContents);

dictFinder = DictionaryFinder(model_name);

chapter = Chapter("Title","Dictionaries");
while hasNext(dictFinder)
    dict = next(dictFinder);
    sect = Section("Title",dict.Name);
    add(sect,dict);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

**See Also**

`systemcomposer.rptgen.finder.DictionaryResult` | `find` | `hasNext` | `next`

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.DictionaryResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for dictionaries

## Description

Search result object for information about a dictionary in a System Composer architecture model.

The systemcomposer.rptgen.finder.DictionaryResult class is a handle class.

## Creation

`result = DictionaryResult` creates a search result object for a dictionary found by a systemcomposer.rptgen.finder.DictionaryFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.DictionaryFinder class creates objects of this type for each dictionary that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### Name — Name of dictionary

string

Name of dictionary, returned as a string.

Data Types: string

### Type — Type of dictionary

"Model" | "Dictionary"

Type of dictionary, returned as "Model" for model dictionaries or "Dictionary" for reference dictionaries.

Data Types: string

### Interfaces — Interfaces in dictionary

array of strings

Interfaces in dictionary, returned as an array of strings.

Data Types: `string`

**Tag — Tag to associate with result**

`string`

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

## **Version History**

**Introduced in R2022b**

### **See Also**

`systemcomposer.rptgen.finder.DictionaryFinder` | `find` | `hasNext` | `next`

### **Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# systemcomposer.rptgen.finder.FunctionFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find function objects

## Description

The `systemcomposer.rptgen.finder.FunctionFinder` class searches for information about all the functions in a given System Composer software architecture model.

## Creation

`finder = FunctionFinder(Container)` creates a finder that finds all functions in a software architecture model specified by the `Properties` property to search for functions with these properties.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

## Properties

### Container — Architecture model file name

string

Architecture model file name without the `.slx` extension, specified as a string.

Example: `f = FunctionFinder("ArchModel")`

Data Types: string

### ComponentName — Component to find functions in

string

Component to find functions in, specified as a string of the full path.

Example: `f.ComponentName = "mTestModel/Component1"`

### Attributes:

<code>GetAccess</code>	public
<code>SetAccess</code>	public

Data Types: string

### **Properties — Properties of objects to find**

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: char

## **Methods**

### **Public Methods**

<code>find</code>	Find information about function
<code>hasNext</code>	Determine if function search result queue is nonempty
<code>next</code>	Get next function search result

## **Version History**

**Introduced in R2022b**

### **See Also**

`systemcomposer.rptgen.finder.FunctionResult` |  
`systemcomposer.rptgen.report.Function` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### **Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.FunctionResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for functions

## Description

Search result object for information about a function in a System Composer software architecture model.

The systemcomposer.rptgen.finder.FunctionResult class is a handle class.

## Creation

result = FunctionResult creates a search result object for a function found by a systemcomposer.rptgen.finder.FunctionFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.FunctionFinder class creates objects of this type for each function that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### Name — Name of function

string

Name of function, returned as a string.

Data Types: string

### Component — Component

string

Component where function is defined, specified as a string.

Data Types: string

### Parent — Parent architecture of component

string

Parent architecture of component where function is defined, specified as a string.

Data Types: `string`

**Period — Period of function**

`numeric` | `string`

Period of function, specified as a numeric value convertible to a string, or a string of valid MATLAB variables. The `Period` property of aperiodic functions is editable. Editing the `Period` property of a periodic function will result in an error.

**ExecutionOrder — Execution order of functions**

row vector of numeric values

Execution order of functions, specified as a row vector of numeric values.

Example: `[model.Architecture.Functions.ExecutionOrder]`

Data Types: `uint64`

**Tag — Tag to associate with result**

`string`

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

**Methods****Public Methods**

`getReporter` Get function reporter

**Version History**

**Introduced in R2022b**

**See Also**

`systemcomposer.rptgen.finder.FunctionFinder` |  
`systemcomposer.rptgen.report.Function` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.InterfaceFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find interfaces

## Description

The `systemcomposer.rptgen.finder.InterfaceFinder` class searches for information about all the interfaces in a given System Composer architecture model.

## Creation

`finder = InterfaceFinder(Container)` creates a finder that finds all interfaces in a given model that meet the `Filter` property.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

## Properties

### Container – Architecture model file name

string

Architecture model file name without the `.slx` extension, specified as a string.

Example: `f = InterfaceFinder("ArchModel")`

Data Types: string

### SearchIn – Where to find interfaces

"Model" | "Component"

Where to find interfaces, specified as "Model" to find interfaces in the model or "Component" to find all interfaces on the ports of a given component.

### Attributes:

GetAccess	public
SetAccess	public

Data Types: string

**Filter – Filter to find interfaces**

"All" | "InterfaceName" | "ComponentName"

Filter to find interfaces, specified as "All" to find all interfaces associated with the model, "InterfaceName" to find a specific interface, or "ComponentName" to find all interfaces on the ports of a given component.

**Attributes:**

GetAccess	public
SetAccess	public

Data Types: string

**Properties – Properties of objects to find**

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: char

**Methods****Public Methods**

find	Find information about interface
hasNext	Determine if interface search result queue is nonempty
next	Get next interface search result

**Examples****Generate Interface Finder Report**

Use the `InterfaceFinder` and `InterfaceResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="InterfaceFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Interfaces in %s Model',model_name)));
add(rpt,TableOfContents);

intfFinder = InterfaceFinder(model_name);

chapter = Chapter("Title","Interfaces");
while hasNext(intfFinder)
    interface = next(intfFinder);
    sect = Section("Title",interface.InterfaceName);
```

```
        add(sect, interface);  
        add(chapter, sect);  
end
```

```
add(rpt, chapter);  
close(rpt);  
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.InterfaceResult |  
systemcomposer.rptgen.report.Interface | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## systemcomposer.rptgen.finder.InterfaceResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for interfaces

### Description

Search result object for information about an interface in a System Composer architecture model.

The systemcomposer.rptgen.finder.InterfaceResult class is a handle class.

### Creation

`result = InterfaceResult` creates a search result object for an interface found by a systemcomposer.rptgen.finder.InterfaceFinder object.

---

**Note** The `find` method of the systemcomposer.rptgen.finder.InterfaceFinder class creates objects of this type for each interface that it finds. You do not need to create this object yourself.

---

### Properties

#### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

#### InterfaceName — Name of interface

string

Name of interface, returned as a string.

Data Types: string

#### Elements — Elements of interface

structure

Elements of interface, returned as a structure with fields.

For data elements:

- Name, returned as a string.
- Type, returned as a string.



- `Description`, returned as a string.
- `Complexity`, returned as a string.
- `Dimensions`, returned as a string.
- `Maximum`, returned as a string.
- `Minimum`, returned as a string.

For value types:

- `Name`, returned as a string.
- `DataType`, returned as a string.
- `Description`, returned as a string.
- `Complexity`, returned as a string.
- `Dimensions`, returned as a string.
- `Maximum`, returned as a string.
- `Minimum`, returned as a string.

For service interfaces:

- `Name`, returned as a string.
- `FunctionPrototype`, returned as a string.
- `FunctionArgument`, returned as a structure with fields:
  - `Name`, returned as a string.
  - `Type`, returned as a string.
  - `Dimensions`, returned as a string.
  - `Description`, returned as a string.

Data Types: `struct`

### **Ports — Ports information**

`structure`

Ports information, returned as a structure with fields:

- `InterfaceName`
- `PortName`
- `FullPortName`
- `Direction`

Data Types: `struct`

### **Tag — Tag to associate with result**

`string`

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

## Methods

### Public Methods

getReporter    Get interface reporter

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.InterfaceFinder |  
systemcomposer.rptgen.report.Interface | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.ProfileFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find profiles

## Description

The `systemcomposer.rptgen.finder.ProfileFinder` class searches for information about profiles in a given System Composer architecture model.

## Creation

`finder = ProfileFinder(Container)` creates a finder that finds profiles in a given model.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

## Properties

### Container – Profile file name

string

Profile file name without the `.xml` extension, specified as a string.

Example: `f = ProfileFinder("TestProfile")`

Data Types: string

### Properties – Properties of objects to find

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: char

## Methods

### Public Methods

`find` Find information about profile

hasNext Determine if profile search result queue is nonempty  
next Get next profile search result

## Examples

### Generate Profile Finder Report

Use the ProfileFinder and ProfileResult classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ProfileFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Profiles in %s Model',model_name)));
add(rpt,TableOfContents);

profileFinder = ProfileFinder("UAVComponent");

chapter = Chapter("Title","Profiles");
while hasNext(profileFinder)
    profile = next(profileFinder);
    sect = Section("Title",profile.Name);
    add(sect,profile);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.ProfileResult |  
systemcomposer.rptgen.report.Profile | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.ProfileResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for profiles

## Description

Search result object for information about a profile in a System Composer architecture model.

The systemcomposer.rptgen.finder.ProfileResult class is a handle class.

## Creation

result = ProfileResult creates a search result object for a profile found by a systemcomposer.rptgen.finder.ProfileFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.ProfileFinder class creates objects of this type for each profile that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### Name — Name of profile

string

Name of profile, returned as a string.

Data Types: string

### Description — Description of profile

string

Description of profile, returned as a string.

Data Types: string

### Stereotypes — Stereotypes on profile

array of strings

Stereotypes on profile, returned as an array of strings.

Data Types: string

**Tag — Tag to associate with result**

string

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: string

**Methods****Public Methods**

getReporter Get profile reporter

**Version History**

Introduced in R2022b

**See Also**

systemcomposer.rptgen.finder.ProfileFinder |  
systemcomposer.rptgen.report.Profile | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.RequirementLinkFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find requirement links

## Description

The `systemcomposer.rptgen.finder.RequirementLinkFinder` class searches for information about requirement links in a requirement link set.

## Creation

`finder = RequirementLinkFinder(Container)` creates a finder that finds requirement links in a given requirement link set.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

## Properties

### Container — Requirement link set file name

string

Requirement link set file name with the `.slmx` extension, specified as a string.

Example: `f = RequirementLinkFinder("System_Reqs.slmx")`

Data Types: string

### Properties — Properties of objects to find

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain', '5'}`

Data Types: char

## Methods

### Public Methods

`find` Find information about requirement link  
`hasNext` Determine if requirement link search result queue is nonempty  
`next` Get next requirement link search result

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.RequirementLinkResult` |  
`systemcomposer.rptgen.report.RequirementLink` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”



# systemcomposer.rptgen.finder.RequirementLinkResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for requirement links

## Description

Search result object for information about a requirement link in a requirement link set.

The systemcomposer.rptgen.finder.RequirementLinkResult class is a handle class.

## Creation

`result = RequirementLinkResult` creates a search result object for a requirement link found by a systemcomposer.rptgen.finder.RequirementLinkFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.RequirementLinkFinder class creates objects of this type for each requirement link that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### Source — Source of link

string

Source of link, returned as a string.

Data Types: string

### Type — Type of link

string

Type of link, returned as a string.

Data Types: string

### Destination — Destination of link

string

Destination of link, returned as a string.

Data Types: `string`

**Tag — Tag to associate with result**

`string`

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

**Methods****Public Methods**

`getReporter` Get requirement links reporter

**Version History**

**Introduced in R2022b**

**See Also**

`systemcomposer.rptgen.finder.RequirementLinkFinder` |  
`systemcomposer.rptgen.report.RequirementLink` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.RequirementSetFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find requirements

## Description

The `systemcomposer.rptgen.finder.RequirementSetFinder` class searches for information about all requirements in a requirement set.

## Creation

`finder = RequirementSetFinder(Container)` creates a finder that finds requirements in a given requirement set.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

## Properties

### Container — Requirement set

string

Requirement set with the `.slreqx` extension, specified as a string.

Example: `f = RequirementSetFinder("System_Reqs.slreqx")`

Data Types: `string`

### Depth — Level to find requirements

numeric value

Level to find requirements, specified as a numeric value.

### Attributes:

GetAccess	public
SetAccess	public

Data Types: `uint64 | inf`

**Properties — Properties of objects to find**

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: char

**Methods****Public Methods**

`find` Find information about requirement  
`hasNext` Determine if requirement set search result queue is nonempty  
`next` Get next requirement set search result

**Version History**

**Introduced in R2022b**

**See Also**

`systemcomposer.rptgen.finder.RequirementSetResult` |  
`systemcomposer.rptgen.report.RequirementSet` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.RequirementSetResult class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Result

Search result for requirements

## Description

Search result object for information about a requirement in a requirement set.

The systemcomposer.rptgen.finder.RequirementSetResult class is a handle class.

## Creation

result = ComponentResult creates a search result object for a requirement found by a systemcomposer.rptgen.finder.RequirementSetFinder object.

---

**Note** The find method of the systemcomposer.rptgen.finder.RequirementSetFinder class creates objects of this type for each requirement that it finds. You do not need to create this object yourself.

---

## Properties

### Object — Universal unique identifier of result element

string

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: string

### ID — ID of requirement

string

ID of requirement, returned as a string.

Data Types: string

### Summary — Summary of requirement

string

Summary of requirement, returned as a string.

Data Types: string

### Link — Requirement link

string

Requirement link, returned as a string.

Data Types: `string`

**Tag — Tag to associate with result**

`string`

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

**Methods****Public Methods**

`getReporter` Get requirements reporter

**Version History**

**Introduced in R2022b**

**See Also**

`systemcomposer.rptgen.finder.RequirementSetFinder` |  
`systemcomposer.rptgen.report.RequirementSet` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.StereotypeFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find stereotypes

## Description

The `systemcomposer.rptgen.finder.StereotypeFinder` class searches for information about stereotypes in a profile in a given System Composer architecture model.

## Creation

`finder = StereotypeFinder(Container)` creates a finder that finds stereotypes in a profile in a given model.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

## Properties

### Container — Profile file name

string

Profile file name without the `.xml` extension, specified as a string.

Example: `f = StereotypeFinder("TestProfile")`

Data Types: string

### StereotypeName — Stereotype name

string

Stereotype name, specified as a string in the form "`<profile>.<stereotype>`".

Example: `f.StereotypeName = "TestProfile.MechanicalComponent"`

### Attributes:

GetAccess	public
SetAccess	public

Data Types: string

**Properties — Properties of objects to find**

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: `char`

**Methods****Public Methods**

`find` Find information about stereotype  
`hasNext` Determine if stereotype search result queue is nonempty  
`next` Get next stereotype search result

**Examples****Generate Stereotype Finder Report**

Use the `StereotypeFinder` and `StereotypeResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="StereotypeFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Stereotypes in %s Model',model_name)));
add(rpt,TableOfContents);

stereotypeFinder = StereotypeFinder("UAVComponent");
chapter = Chapter("Title","Stereotypes");
while hasNext(stereotypeFinder)
    stereotype = next(stereotypeFinder);
    sect = Section("Title",stereotype.Name);
    add(sect,stereotype);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

**Version History**

**Introduced in R2022b**



## See Also

[systemcomposer.rptgen.finder.StereotypeResult](#) |  
[systemcomposer.rptgen.report.Stereotype](#) | [find](#) | [hasNext](#) | [next](#) | [getReporter](#) |  
[createTemplate](#) | [customizeReporter](#) | [getClassFolder](#)

## Topics

[“System Composer Report Generation for System Architectures”](#)  
[“System Composer Report Generation for Software Architectures”](#)

## **systemcomposer.rptgen.finder.StereotypeResult class**

**Package:** `systemcomposer.rptgen.finder`

**Superclasses:** `mlreportgen.finder.Result`

Search result for stereotypes

### **Description**

Search result object for information about a stereotype in a profile in a given System Composer architecture model.

The `systemcomposer.rptgen.finder.StereotypeResult` class is a handle class.

### **Creation**

`result = StereotypeResult` creates a search result object for a stereotype found by a `systemcomposer.rptgen.finder.StereotypeFinder` object.

---

**Note** The `find` method of the `systemcomposer.rptgen.finder.StereotypeFinder` class creates objects of this type for each stereotype that it finds. You do not need to create this object yourself.

---

### **Properties**

#### **Object — Universal unique identifier of result element**

`string`

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: `string`

#### **Name — Name of stereotype**

`string`

Name of stereotype, specified as a string. This property must be a valid MATLAB identifier.

Example: `"HardwareComponent"`

Data Types: `string`

#### **Icon — Icon for stereotype**

reporter object

Icon for stereotype, specified as a `mlreportgen.dom.Image` object.

#### **Parent — Stereotype from which stereotype inherits properties**

stereotype object

Stereotype from which stereotype inherits properties, specified as a `systemcomposer.profile.Stereotype` object.

### **Description — Description text for stereotype**

string

Description text for stereotype, specified as a string.

Data Types: string

### **AppliesTo — Element type to which stereotype can be applied**

"" (default) | "Component" | "Port" | "Connector" | "Interface" | "Function" | "Requirement" | "Link"

Element type to which stereotype can be applied, specified as one of these options:

- "" to apply stereotype to all element types
- "Component"
- "Port"
- "Connector"
- "Interface"
- "Function", which is only available for software architectures
- "Requirement", to be used with Requirements Toolbox
- "Link", to be used with Requirements Toolbox

Data Types: string

### **Properties — Properties**

structure

Properties contained in stereotype and inherited from the stereotype base hierarchy, returned as a structure with fields:

- Name, returned as a string.
- Type, returned as a string.
- Index, returned as an integer.
- Unit, returned as a string.
- DefaultValue, returned as a string.

Data Types: struct

### **Tag — Tag to associate with result**

string

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: string

## Methods

### Public Methods

getReporter    Get stereotype reporter

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.StereotypeFinder |  
systemcomposer.rptgen.report.Stereotype | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.finder.ViewFinder class

**Package:** systemcomposer.rptgen.finder

**Superclasses:** mlreportgen.finder.Finder

Find views

## Description

The `systemcomposer.rptgen.finder.ViewFinder` class searches for information about all the views in a given System Composer architecture model.

## Creation

`finder = ViewFinder(Container)` creates a finder that finds all views in a given model.

---

**Note** This finder provides two ways to get search results:

- To return the search results as an array, use the `find` method. Add the results directly to a report or process the results in a `for` loop.
- To iterate through the results one at a time, use the `hasNext` and `next` methods in a `while` loop.

Neither option has a performance advantage.

---

## Properties

### Container — Architecture model file name

string

Architecture model file name without the `.slx` extension, specified as a string.

Example: `f = ViewFinder("ArchModel")`

Data Types: string

### DiagramType — Type of view

"Default" | "Component Diagram" | "Component Hierarchy"

Type of view, specified as "Default" to display what the view was saved in, "Component Diagram" for component diagram, and "Component Hierarchy" for component hierarchy.

### Attributes:

GetAccess	public
SetAccess	public

Data Types: string

### Properties — Properties of objects to find

cell array of name-value arguments

Properties of objects to find, specified as a cell array of name-value arguments. The finder returns only objects that have the specified properties with the specified values.

Example: `f.Properties = {'Gain','5'}`

Data Types: `char`

## Methods

### Public Methods

`find` Find information about view  
`hasNext` Determine if view search result queue is nonempty  
`next` Get next view search result

## Examples

### Generate View Finder Report

Use the `ViewFinder` and `ViewResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

sckKeylessEntrySystem
model_name = "KeylessEntryArchitecture";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ViewFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Views in %s Model',model_name)));
add(rpt,TableOfContents);

viewFinder = ViewFinder(model_name);

chapter = Chapter("Title","Views");
while hasNext(viewFinder)
    view = next(viewFinder);
    sect = Section("Title",view.Name);
    add(sect,view);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

**Introduced in R2022b**

## See Also

systemcomposer.rptgen.finder.ViewResult | systemcomposer.rptgen.report.View |  
find | hasNext | next | getReporter | createTemplate | customizeReporter |  
getClassFolder

## Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## **systemcomposer.rptgen.finder.ViewResult class**

**Package:** `systemcomposer.rptgen.finder`

**Superclasses:** `mlreportgen.finder.Result`

Search result for views

### **Description**

Search result object for information about a view in a System Composer architecture model.

The `systemcomposer.rptgen.finder.ViewResult` class is a `handle` class.

### **Creation**

`result = ViewResult` creates a search result object for a view found by a `systemcomposer.rptgen.finder.ViewFinder` object.

---

**Note** The `find` method of the `systemcomposer.rptgen.finder.ViewFinder` class creates objects of this type for each view that it finds. You do not need to create this object yourself.

---

### **Properties**

#### **Object — Universal unique identifier of result element**

`string`

Universal unique identifier (UUID) of result element, returned as a string.

Data Types: `string`

#### **Name — Name of view**

`string`

Name of view, specified as a string.

Example: "NewView"

Data Types: `string`

#### **Description — Description of view**

`string`

Description of view, specified as a string.

Data Types: `string`

#### **Select — Selection query**

`string`

Selection query associated with view, specified as a string.

Data Types: `string`



**GroupBy — Grouping criteria**

string array of properties

Grouping criteria, specified as a string array of properties in the form "`<profile>.<stereotype>.<property>`".

Example:

```
["AutoProfile.MechanicalComponent.mass", "AutoProfile.MechanicalComponent.cost"]
```

**Elements — Elements in view**

array of component objects

Elements in view, returned as an array of `systemcomposer.arch.Component` objects.

**SubGroups — Subgroups in view**

array of element group objects

Subgroups in view, returned as an array of `systemcomposer.view.ElementGroup` objects.

Data Types: `char` | `string`

**Snapshot — Snapshot of view**

reporter object

Custom snapshot reporter, specified as a `mlreportgen.report.FormalImage` object.

**Color — Color of view**

string

Color of view, specified as a string. The color can be the name "blue", "black", or "green", or it can be an RGB value encoded in a hexadecimal string: "#FF00FF" or "#DDDDDD". An invalid color results in an error.

**Tag — Tag to associate with result**

string

Tag to associate with result, specified as a string. This property allows you to attach additional information to a result. You can set this property to any value that meets your requirements.

Data Types: `string`

**Methods****Public Methods**

`getReporter` Get view reporter

**Version History**

Introduced in R2022b

**See Also**

`systemcomposer.rptgen.finder.ViewFinder` | `systemcomposer.rptgen.report.View` | `find` | `hasNext` | `next` | `getReporter` | `createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.report.AllocationList class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Allocation list reporter

## Description

Create a reporter that reports on all the components to and from which a particular component has been allocated in a System Composer architecture model.

The systemcomposer.rptgen.report.AllocationList class is a handle class.

## Creation

`reporter = AllocationList("Source", result)` creates a reporter that reports on allocations around the component defined by the ComponentName property using a systemcomposer.rptgen.finder.AllocationListResult object.

## Properties

### Source — Allocation list result

allocation list result object

Allocation list result, specified as a systemcomposer.rptgen.finder.AllocationListResult object.

### AllocatedFrom — Component from which specified component has been allocated

ordered list object

Component from which specified component has been allocated, specified as an mlreportgen.dom.OrderedList object.

### AllocatedTo — Component to which specified component has been allocated

ordered list object

Component to which specified component has been allocated, specified as an mlreportgen.dom.OrderedList object..

### IncludeAllocatedFrom — Whether to report on allocated-from list

true or 1 | false or 0

Whether to report on allocated-from list, specified as a logical.

Data Types: logical

### IncludeAllocatedTo — Whether to report on allocated-to list

true or 1 | false or 0

Whether to report on allocated-to list, specified as a logical.

Data Types: `logical`

### TemplateSrc — Source of template for this reporter

[] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft® Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

### TemplateName — Name of template for this reporter

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

### LinkTarget — Hyperlink target for this reporter

[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

## Methods

### Public Methods

`createTemplate` Create allocation list template  
`customizeReporter` Create custom allocation list reporter class  
`getClassFolder` Allocation list class definition file location

### Inherited Methods

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

## Examples

### Generate AllocationList Result Report

Use the `AllocationListFinder` and `AllocationListResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationListResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocations"));
add(rpt,TableOfContents);

allocationListFinder = AllocationListFinder("AllocationSet.mldatx");
allocationListFinder.ComponentName = "mTestModel/Component1";
chapter = Chapter("Title",allocationListFinder.ComponentName);
result = find(allocationListFinder);
reporter = getReporter(result);

add(rpt,chapter);
append(rpt,reporter);
close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationListFinder |  
systemcomposer.rptgen.finder.AllocationListResult | find | next | hasNext |  
getReporter | createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## systemcomposer.rptgen.report.AllocationSet class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Allocation set reporter

### Description

Create a reporter that reports on an allocation set file used between System Composer models.

The systemcomposer.rptgen.report.AllocationSet class is a handle class.

### Creation

reporter = AllocationSet("Source", result) creates a reporter that reports on an allocation set using a systemcomposer.rptgen.finder.AllocationSetResult object specified by "Source".

### Properties

#### Source — Allocation set result

allocation set result object

Allocation set result, specified as a systemcomposer.rptgen.finder.AllocationSetResult object.

#### Summary — Custom summary reporter

reporter object

Custom summary reporter, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

#### Scenario — Scenarios in allocation set

reporter object

Scenarios in allocation set, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

#### IncludeSummary — Whether to include summary table

true or 1 | false or 0

Whether to include summary table, specified as a logical.

Data Types: logical

#### IncludeScenario — Whether to include allocation scenario table

true or 1 | false or 0

Whether to include allocation scenario table, specified as a logical.

Data Types: `logical`

### TemplateSrc — Source of template for this reporter

[ ] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

### TemplateName — Name of template for this reporter

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

### LinkTarget — Hyperlink target for this reporter

[ ] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

## Methods

### Public Methods

`createTemplate` Create allocation set template  
`customizeReporter` Create custom allocation set reporter class  
`getClassFolder` Allocation set class definition file location

### Inherited Methods

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

## Examples

### Generate AllocationSet Result Report

Use the `AllocationSetFinder` and `AllocationSetResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationSetResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocation Sets"));
add(rpt,TableOfContents);
chapter = Chapter("Title","Allocation Sets");

allocationSetFinder = AllocationSetFinder("AllocationSet.mldatx");
result = find(allocationSetFinder);
reporter = getReporter(result);

add(rpt,chapter);
append(rpt,reporter);
close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationSetFinder |  
systemcomposer.rptgen.finder.AllocationSetResult | find | hasNext | next |  
getReporter | createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”



# systemcomposer.rptgen.report.Component class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Component reporter

## Description

Create a reporter that reports on all components in a System Composer architecture model.

The systemcomposer.rptgen.report.Component class is a handle class.

## Creation

reporter = Component("Source", result) creates a reporter that reports on a component using a systemcomposer.rptgen.finder.ComponentResult object.

## Properties

### Source — Component result

component result object

Component result, specified as a systemcomposer.rptgen.finder.ComponentResult object.

### Snapshot — Custom snapshot reporter

reporter object

Custom snapshot reporter, specified as a reporter object. The default value is the slreportgen.report.Diagram reporter.

### Properties — Custom properties reporter

reporter object

Custom properties reporter, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

### Stereotypes — Custom properties reporter for stereotypes on component

reporter object

Custom properties reporter for stereotypes on component, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

### Ports — Custom properties reporter for ports on component

reporter object

Custom properties reporter for ports on component, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

### Functions — Custom properties reporter for functions on software component

reporter object

Custom properties reporter for functions on software component, specified as a reporter object. The default value is the `mreportgen.report.BaseTable` reporter.

**Children — Child components**

array of component reporter objects

Child components, specified as an array of `systemcomposer.rptgen.report.Component` objects.

Data Types: `string`

**IncludeSnapshot — Whether to include snapshot table**

`true` or `1` | `false` or `0`

Whether to include snapshot table, specified as a logical.

Data Types: `logical`

**IncludeProperties — Whether to include properties table**

`true` or `1` | `false` or `0`

Whether to include properties table, specified as a logical.

Data Types: `logical`

**IncludeStereotypes — Whether to include stereotypes table**

`true` or `1` | `false` or `0`

Whether to include stereotypes table, specified as a logical.

Data Types: `logical`

**IncludePorts — Whether to include ports table**

`true` or `1` | `false` or `0`

Whether to include ports table, specified as a logical.

Data Types: `logical`

**IncludeFunctions — Whether to include functions table**

`true` or `1` | `false` or `0`

Whether to include functions table, specified as a logical.

Data Types: `logical`

**TemplateSrc — Source of template for this reporter**

`[]` (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

### TemplateName — Name of template for this reporter

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

### LinkTarget — Hyperlink target for this reporter

[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

## Methods

### Public Methods

`createTemplate` Create component template  
`customizeReporter` Create custom component reporter class  
`getClassFolder` Component class definition file location

### Inherited Methods

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

## Examples

### Generate Component Result Report

Use the `ComponentFinder` and `ComponentResult` classes to generate a report.

```
import systemcomposer.rptgen.finder.*
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.query.*

rpt = slreportgen.report.Report(output="ComponentResultReport", ...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Components"));
add(rpt,TableOfContents);
chapter = Chapter("Title","Components");

componentFinder = ComponentFinder("mTestModel");
componentFinder.Query = AnyComponent;
result = find(componentFinder);
```

```
for i = result
    reporter = getReporter(i);
    reporter.IncludeProperties = false;
    reporter.IncludeSnapshot = false;
    add(chapter, reporter);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.ComponentFinder |  
systemcomposer.rptgen.finder.ComponentResult | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.report.Connector class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Connector reporter

## Description

Create a reporter that reports on all connectors in a System Composer architecture model.

The systemcomposer.rptgen.report.Connector class is a handle class.

## Creation

reporter = Connector("Source", result) creates a reporter that reports on a connector using a systemcomposer.rptgen.finder.ConnectorResult object.

## Properties

### Source — Connector result

connector result object

Connector result, specified as a systemcomposer.rptgen.finder.ConnectorResult object.

### Summary — Custom summary reporter

reporter object

Custom summary reporter, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

### TemplateSrc — Source of template for this reporter

[] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, TemplateSrc must be a Word reporter template. If the TemplateSrc property is empty, this reporter uses the default reporter template for the output type of the report.

### TemplateName — Name of template for this reporter

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

### LinkTarget — Hyperlink target for this reporter

[ ] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

## Methods

### Public Methods

`createTemplate` Create connector template  
`customizeReporter` Create custom connector reporter class  
`getClassFolder` Connector class definition file location

### Inherited Methods

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

## Examples

### Generate Connector Result Report

Use the `ConnectorFinder` and `ConnectorResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ConnectorResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Connectors in %s Model',model_name)));
add(rpt,TableOfContents);

connectorFinder = ConnectorFinder(model_name);
connectorFinder.Filter = "Component";
connectorFinder.ComponentName = "scExampleSmallUAVModel/Flight Support Components";
chapter = Chapter("Title","Connectors");
result = find(connectorFinder);
add(rpt,chapter);

for r = result
    reporter = getReporter(r);
    append(rpt,reporter);
end
```

```
close(rpt);  
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ConnectorFinder |  
systemcomposer.rptgen.finder.ConnectorResult | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## systemcomposer.rptgen.report.DependencyGraph class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Dependency graph reporter

### Description

Create a reporter that reports on a dependency graph for a System Composer architecture model artifact.

The `systemcomposer.rptgen.report.DependencyGraph` class is a handle class.

### Creation

`reporter = DependencyGraph("Source", fullpath)` creates a reporter that reports on a dependency graph using the full path of the artifact.

### Properties

#### Source — Full path to artifact

string

Full path to artifact, specified as a string.

Data Types: string

#### Layout — Alignment of dependency graph

"Vertical" (default) | "Horizontal"

Alignment of dependency graph, specified as "Vertical" for a vertically aligned dependency graph or "Horizontal" for a horizontally aligned dependency graph.

Data Types: string

#### Snapshot — Custom snapshot reporter

reporter object

Custom snapshot reporter, specified as a reporter object. The default value is the `slreportgen.report.Diagram` reporter.

#### TemplateSrc — Source of template for this reporter

[] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter



- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

### TemplateName — Name of template for this reporter

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

### LinkTarget — Hyperlink target for this reporter

[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

## Methods

### Public Methods

<code>createTemplate</code>	Create dependency graph template
<code>customizeReporter</code>	Create custom dependency graph reporter class
<code>getClassFolder</code>	Dependency graph class definition file location

### Inherited Methods

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

## Version History

Introduced in R2022b

### See Also

`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”  
 “System Composer Report Generation for Software Architectures”

## systemcomposer.rptgen.report.Function class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Function reporter

### Description

Create a reporter that reports on all functions in a System Composer software architecture model.

The `systemcomposer.rptgen.report.Function` class is a `handle` class.

### Creation

`reporter = Function("Source", result)` creates a reporter that reports on a function using a `systemcomposer.rptgen.finder.FunctionResult` object.

### Properties

#### Source — Function result

function result object

Function result, specified as a `systemcomposer.rptgen.finder.FunctionResult` object.

#### Summary — Custom summary reporter

reporter object

Custom summary reporter, specified as a reporter object. The default value is the `mlreportgen.report.BaseTable` reporter.

#### TemplateSrc — Source of template for this reporter

[ ] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

#### TemplateName — Name of template for this reporter

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

### LinkTarget — Hyperlink target for this reporter

[ ] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

## Methods

### Public Methods

`createTemplate` Create function template  
`customizeReporter` Create custom function reporter class  
`getClassFolder` Function class definition file location

### Inherited Methods

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.finder.FunctionFinder` |  
`systemcomposer.rptgen.finder.FunctionResult` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## **systemcomposer.rptgen.report.Interface class**

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Interface reporter

### **Description**

Create a reporter that reports on interfaces in a System Composer architecture model.

The systemcomposer.rptgen.report.Interface class is a handle class.

### **Creation**

reporter = Interface("Source", result) creates a reporter that reports on interfaces in a model using a systemcomposer.rptgen.finder.InterfaceResult object.

### **Properties**

#### **Source — Interface result**

interface result object

Interface result, specified as a systemcomposer.rptgen.finder.InterfaceResult object.

#### **Elements — Elements in interface of component**

reporter object

Elements in interface of component, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

#### **PortsUsage — Ports on which interface is present**

reporter object

Ports on which interface is present, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

#### **IncludeElements — Whether to report on elements table**

true or 1 | false or 0

Whether to report on allocated from list, specified as a logical.

Data Types: logical

#### **IncludePortsUsage — Whether to report on ports usage table**

true or 1 | false or 0

Whether to report on allocated to list, specified as a logical.

Data Types: logical

**TemplateSrc — Source of template for this reporter**

[] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

**TemplateName — Name of template for this reporter**

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

**LinkTarget — Hyperlink target for this reporter**

[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

**Methods****Public Methods**

`createTemplate`      Create interface template  
`customizeReporter`    Create custom interface reporter class  
`getClassFolder`      Interface class definition file location

**Inherited Methods**

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

**Version History**

Introduced in R2022b

**See Also**

`systemcomposer.rptgen.finder.InterfaceFinder` |  
`systemcomposer.rptgen.finder.InterfaceResult` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## systemcomposer.rptgen.report.Profile class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Profile reporter

### Description

Create a reporter that reports on profile files that can be used with a System Composer architecture model.

The systemcomposer.rptgen.report.Profile class is a handle class.

### Creation

reporter = Profile("Source", result) creates a reporter that reports on profiles in a model using a systemcomposer.rptgen.finder.ProfileResult object.

### Properties

#### Source — Profile result

profile result object

Profile result, specified as a systemcomposer.rptgen.finder.ProfileResult object.

#### Summary — Custom summary reporter

reporter object

Custom summary reporter, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

#### TemplateSrc — Source of template for this reporter

[] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, TemplateSrc must be a Word reporter template. If the TemplateSrc property is empty, this reporter uses the default reporter template for the output type of the report.

**TemplateName — Name of template for this reporter**

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

**LinkTarget — Hyperlink target for this reporter**[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

**Methods****Public Methods**

`createTemplate`      Create profile template  
`customizeReporter`    Create custom profile reporter class  
`getClassFolder`      Profile class definition file location

**Inherited Methods**

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

**Version History**

Introduced in R2022b

**See Also**

`systemcomposer.rptgen.finder.ProfileFinder` |  
`systemcomposer.rptgen.finder.ProfileResult` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”



# systemcomposer.rptgen.report.RequirementLink class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Requirement link reporter

## Description

Create a reporter that reports on all requirement links in requirement link set.

The systemcomposer.rptgen.report.RequirementLink class is a handle class.

## Creation

reporter = RequirementLink("Source", result) creates a reporter that reports on a requirement link set using a systemcomposer.rptgen.finder.RequirementLinkResult object.

## Properties

### Source — Requirement link result

requirement link result object

Requirement link result, specified as a systemcomposer.rptgen.finder.RequirementLinkResult object.

### Summary — Custom summary reporter

reporter object

Custom summary reporter, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

### TemplateSrc — Source of template for this reporter

[] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, TemplateSrc must be a Word reporter template. If the TemplateSrc property is empty, this reporter uses the default reporter template for the output type of the report.

**TemplateName — Name of template for this reporter**

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

**LinkTarget — Hyperlink target for this reporter**[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

**Methods****Public Methods**

`createTemplate` Create requirement link template  
`customizeReporter` Create custom requirement link reporter class  
`getClassFolder` Requirement link class definition file location

**Inherited Methods**

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

**Version History**

Introduced in R2022b

**See Also**

`systemcomposer.rptgen.finder.RequirementLinkFinder` |  
`systemcomposer.rptgen.finder.RequirementLinkResult` | `find` | `hasNext` | `next` |  
`getReporter` | `createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.report.RequirementSet class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Requirement set reporter

## Description

Create a reporter that reports on all requirements in a requirement set.

The systemcomposer.rptgen.report.RequirementSet class is a handle class.

## Creation

`reporter = RequirementSet("Source", result)` creates a reporter that reports on a requirement set using a `systemcomposer.rptgen.finder.RequirementSetResult` object.

## Properties

### Source — Requirement set result

requirement set result object

Requirement set result, specified as a `systemcomposer.rptgen.finder.RequirementSetResult` object.

### Properties — Custom properties reporter

reporter object

Custom properties reporter, specified as a reporter object. The default value is the `mlreportgen.report.BaseTable` reporter.

### TemplateSrc — Source of template for this reporter

[] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

**TemplateName — Name of template for this reporter**

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

**LinkTarget — Hyperlink target for this reporter**[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

**Methods****Public Methods**

`createTemplate` Create requirement set template  
`customizeReporter` Create custom requirement set reporter class  
`getClassFolder` Requirement set class definition file location

**Inherited Methods**

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

**Version History**

Introduced in R2022b

**See Also**

`systemcomposer.rptgen.finder.RequirementSetFinder` |  
`systemcomposer.rptgen.finder.RequirementSetResult` | `find` | `hasNext` | `next` |  
`getReporter` | `createTemplate` | `customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.report.SequenceDiagram class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Sequence diagram reporter

## Description

Create a reporter that reports on a sequence diagram in a System Composer architecture model.

The systemcomposer.rptgen.report.SequenceDiagram class is a handle class.

## Creation

`reporter = SequenceDiagram("Name", name, "ModelName", model)` creates a reporter that reports on a sequence diagram using the name and model name.

## Properties

### Name — Name of sequence diagram

string

Name of sequence diagram, specified as a string.

Data Types: string

### ModelName — Architecture model file name

string

Architecture model file name without the .slx extension, specified as a string.

Data Types: string

### Snapshot — Custom snapshot reporter

reporter object

Custom snapshot reporter, specified as a reporter object. The default value is the slreportgen.report.Diagram reporter.

### TemplateSrc — Source of template for this reporter

[ ] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter

- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

#### **TemplateName — Name of template for this reporter**

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

#### **LinkTarget — Hyperlink target for this reporter**

[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

## Methods

### Public Methods

<code>createTemplate</code>	Create sequence diagram template
<code>customizeReporter</code>	Create custom sequence diagram reporter class
<code>getClassFolder</code>	Sequence diagram class definition file location

### Inherited Methods

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

## Version History

Introduced in R2022b

### See Also

`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”  
 “System Composer Report Generation for Software Architectures”

# systemcomposer.rptgen.report.Stereotype class

**Package:** systemcomposer.rptgen.report

**Superclasses:** slreportgen.report.Reporter

Stereotype reporter

## Description

Create a reporter that reports on all stereotypes in a profile that can be used with a System Composer architecture model.

The systemcomposer.rptgen.report.Stereotype class is a handle class.

## Creation

reporter = Stereotype("Source", result) creates a reporter that reports on a stereotype using a systemcomposer.rptgen.finder.StereotypeResult object.

## Properties

### Source — Stereotype result

stereotype result object

Stereotype result, specified as a systemcomposer.rptgen.finder.StereotypeResult object.

### Summary — Custom summary reporter

reporter object

Custom summary reporter, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

### Properties — Custom properties reporter

reporter object

Custom properties reporter, specified as a reporter object. The default value is the mlreportgen.report.BaseTable reporter.

### IncludeSummary — Whether to include summary table

true or 1 | false or 0

Whether to include summary table, specified as a logical.

Data Types: logical

### IncludeProperties — Whether to include properties table

true or 1 | false or 0

Whether to include properties table, specified as a logical.

Data Types: logical

**TemplateSrc — Source of template for this reporter**

[] (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

**TemplateName — Name of template for this reporter**

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

**LinkTarget — Hyperlink target for this reporter**[] (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

**Methods****Public Methods**

`createTemplate` Create stereotype template  
`customizeReporter` Create custom stereotype reporter class  
`getClassFolder` Stereotype class definition file location

**Inherited Methods**

<code>copy</code>	Create copy of a Simulink reporter object and make deep copies of certain property values
<code>getImpl</code>	Get implementation of reporter

**Version History**

Introduced in R2022b

**See Also**

`systemcomposer.rptgen.finder.StereotypeFinder` |  
`systemcomposer.rptgen.finder.StereotypeResult` | `find` | `hasNext` | `next` | `getReporter`  
| `createTemplate` | `customizeReporter` | `getClassFolder`



**Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## **systemcomposer.rptgen.report.View class**

**Package:** `systemcomposer.rptgen.report`

**Superclasses:** `slreportgen.report.Reporter`

View reporter

### **Description**

Create a reporter that reports on all views in a System Composer architecture model.

The `systemcomposer.rptgen.report.View` class is a `handle` class.

### **Creation**

`reporter = View("Source", result)` creates a reporter that reports on a view using a `systemcomposer.rptgen.finder.ViewResult` object.

### **Properties**

#### **Source — View result**

view result object

View result, specified as a `systemcomposer.rptgen.finder.ViewResult` object.

#### **Snapshot — Custom snapshot reporter**

reporter object

Custom snapshot reporter, specified as a reporter object. The default value is the `slreportgen.report.Diagram` reporter.

#### **Elements — Elements present in view**

reporter object

Elements present in view, specified as a reporter object. The default value is the `mlreportgen.report.BaseTable` reporter.

#### **Properties — Custom properties reporter**

reporter object

Custom properties reporter, specified as a reporter object. The default value is the `mlreportgen.report.BaseTable` reporter.

#### **SubGroups — Subgroups of view**

reporter object

Subgroups of view, specified as a reporter object. The default value is the `mlreportgen.report.BaseTable` reporter.

#### **IncludeElements — Whether to include elements table**

true or 1 | false or 0

Whether to include elements table, specified as a logical.

Data Types: `logical`

### **IncludeProperties — Whether to include properties table**

`true` or `1` | `false` or `0`

Whether to include properties table, specified as a logical.

Data Types: `logical`

### **IncludeSubGroups — Whether to include subgroups table**

`true` or `1` | `false` or `0`

Whether to include subgroups table, specified as a logical.

Data Types: `logical`

### **TemplateSrc — Source of template for this reporter**

`[]` (default) | character vector | string scalar | reporter or report | DOM document or document part

Source of the template for this reporter, specified as one of these options:

- Character vector or string scalar that specifies the path of the file that contains the template for this reporter
- Reporter or report whose template is used for this reporter or whose template library contains the template for this reporter
- DOM document or document part whose template is used for this reporter or whose template library contains the template for this reporter

The specified template must be the same type as the report to which this reporter is appended. For example, for a Microsoft Word report, `TemplateSrc` must be a Word reporter template. If the `TemplateSrc` property is empty, this reporter uses the default reporter template for the output type of the report.

### **TemplateName — Name of template for this reporter**

character vector | string scalar

Name of template for this reporter, specified as a character vector or string scalar. The template for this reporter must be in the template library of the template source (`TemplateSrc`) for this reporter.

### **LinkTarget — Hyperlink target for this reporter**

`[]` (default) | character vector | string scalar | `mlreportgen.dom.LinkTarget` object

Hyperlink target for this reporter, specified as a character vector or string scalar that specifies the link target ID or as an `mlreportgen.dom.LinkTarget` object. A character vector or string scalar value is converted to a `LinkTarget` object. The link target immediately precedes the content of this reporter in the output report.

## **Methods**

### **Public Methods**

`createTemplate`      Create view template  
`customizeReporter`    Create custom view reporter class

getClassFolder      View class definition file location

### **Inherited Methods**

copy	Create copy of a Simulink reporter object and make deep copies of certain property values
getImpl	Get implementation of reporter

## **Version History**

**Introduced in R2022b**

### **See Also**

systemcomposer.rptgen.finder.ViewFinder |  
systemcomposer.rptgen.finder.ViewResult | find | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### **Topics**

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# Functions

---

## addChoice

**Package:** `systemcomposer.arch`

Add variant choices to variant component

### Syntax

```
compList = addChoice(variantComponent,choices)
compList = addChoice(variantComponent,choices,labels)
```

### Description

`compList = addChoice(variantComponent,choices)` creates variant choices specified in `choices` in the specified variant component and returns their handles.

`compList = addChoice(variantComponent,choices,labels)` creates variant choices specified in `choices` with labels `labels` in the specified variant component and returns their handles.

### Examples

#### Add Variant Choices

Create a model, get the root architecture, create one variant component, and add two choices for the variant component.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
variant = addVariantComponent(arch,"Component1");
compList = addChoice(variant,["Choice1","Choice2"]);
```

### Input Arguments

#### **variantComponent** – Variant component

variant component object

Variant component, specified as a `systemcomposer.arch.VariantComponent` object.

#### **choices** – Variant choice names

cell array of character vectors | array of strings

Variant choice names, specified as a cell array of character vectors or an array of strings. The length of `choices` must be the same as `labels`.

Data Types: `char` | `string`

#### **labels** – Variant choice labels

cell array of character vectors | array of strings

Variant choice labels, specified as a cell array of character vectors or an array of strings. The length of labels must be the same as choices.

Data Types: `char` | `string`

## Output Arguments

### **compList** — Created components

array of components

Created components, returned as an array of `systemcomposer.arch.Component` objects. This array is the same size as choices and labels.

## More About

### Definitions

Term	Definition	Application	More Information
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## Version History

Introduced in R2019a

### See Also

`getActiveChoice` | `getChoices` | `makeVariant` | `addVariantComponent` | Variant Component

### Topics

"Create Variants"

# addComponent

**Package:** systemcomposer.arch

Add components to architecture

## Syntax

```
components = addComponent(arch, compNames)
components = addComponent(arch, compNames, stereotypes)
```

## Description

`components = addComponent(arch, compNames)` adds a set of components specified by the names `compNames`.

To remove a component, use the `destroy` function.

`components = addComponent(arch, compNames, stereotypes)` applies stereotypes specified in `stereotypes` to the new components.

## Examples

### Create Model with Two Components

Create a model, get the root architecture, and create components. Arrange the layout to view both components.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
names = ["Component1", "Component2"];
comps = addComponent(arch, names);
Simulink.BlockDiagram.arrangeSystem("archModel");
```

## Input Arguments

### arch — Architecture

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### compNames — Names of components

cell array of character vectors | array of strings

Name of components, specified as a cell array of character vectors or an array of strings. The length of `compNames` must be the same as `stereotypes`.

Data Types: `char` | `string`

### stereotypes — Stereotypes to apply to components

cell array of character vectors | array of strings



Stereotypes to apply to components, specified as a cell array of character vectors or an array of strings. Each element is the qualified stereotype name for the corresponding component in the form "<profile>.<stereotype>".

Data Types: char | string

## Output Arguments

### **components** — Created components

array of component objects

Created components, returned as an array of `systemcomposer.arch.Component` objects.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

addPort | connect | Component

### Topics

“Components”

# addComponent

**Package:** `systemcomposer.view`

(Removed) Add component to view given path

---

**Note** The `addComponent` function has been removed. You can create a view using the `createView` function and add a component using the `addElement` function. For further details, see “Compatibility Considerations”.

---

## Syntax

```
viewComp = addComponent(object, compPath)
```

## Description

`viewComp = addComponent(object, compPath)` adds the component with the specified path.

`addComponent` is a method for the class `systemcomposer.view.ViewArchitecture`.

## Examples

### Add Component to View

Create a model, extract its architecture, and add three components.

```
model = systemcomposer.createModel('mobileRobotAPI');
arch = model.Architecture;
components = addComponent(arch, {'Sensor', 'Planning', 'Motion'});
```

Create a view architecture, a view component, and add a component. Open the **Architecture Views Gallery** to view the component.

```
view = model.createViewArchitecture('NewView');
viewComp = fobSupplierView.createViewComponent('ViewComp');
viewComp.Architecture.addComponent('mobileRobotAPI/Motion');
openViews(model);
```

## Input Arguments

### **object** — View architecture

view architecture object

View architecture, specified as a `systemcomposer.view.ViewArchitecture` object.

### **compPath** — Path to component

character vector

Path to component, including the name of the top-level model, specified as a character vector.

Example: 'mobileRobotAPI/Motion'

Data Types: char

## Output Arguments

### **viewComp — View component**

view component object

View component, returned as a `systemcomposer.view.ViewComponent` object.

## Version History

### **Introduced in R2019b**

### **R2021a: addComponent function has been removed**

*Errors starting in R2021a*

The `addComponent` function is removed in R2021a with the introduction of new views APIs. For more information on how to create and edit a view programmatically, see “Create Architectural Views Programmatically”.

## See Also

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup`

## Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# addElement

**Package:** systemcomposer.interface

Add element

## Syntax

```
element = addElement(interface,name)
element = addElement(interface,name,Name,Value)
```

## Description

`element = addElement(interface,name)` adds an element to an interface with default properties.

To remove an element from an interface, use the `removeElement` function.

`element = addElement(interface,name,Name,Value)` sets the properties of the element using name-value arguments.

## Examples

### Add Data Interface and Data Element

Create a new model `newModel`. Add a data interface `newInterface` to the interface dictionary of the model. Then, add a data element `newElement` with data type `double`.

```
arch = systemcomposer.createModel("newModel",true);
interface = addInterface(arch.InterfaceDictionary,"newInterface");
element = addElement(interface,"newElement",DataType="double")

element =
```

DataElement with properties:

```
Interface: [1x1 systemcomposer.interface.DataInterface]
Name: 'newElement'
Type: [1x1 systemcomposer.ValueType]
UUID: '2d267175-33c2-43a9-be41-albe2774a3cf'
ExternalUID: ''
```

### Add Physical Interface and Physical Element

Create a new model named `'newModel'`. Add a physical interface `'newInterface'` to the interface dictionary of the model. Then, add a physical element `'newElement'` with type `'electrical.electrical'`. Change the physical domain type to `'electrical.six_phase'`.

```
arch = systemcomposer.createModel('newModel',true);
interface = addPhysicalInterface(arch.InterfaceDictionary,'newInterface');
```

```
element = addElement(interface,'newElement','Type','electrical.electrical');
element.Type = 'electrical.six_phase';
element
```

```
element =
```

```
PhysicalElement with properties:
```

```
    Name: 'newElement'
    Type: [1x1 systemcomposer.interface.PhysicalDomain]
    Interface: [1x1 systemcomposer.interface.PhysicalInterface]
    UUID: '32e4c51e-e567-42f1-b44a-2d2fcdabb5c25'
    ExternalUUID: ''
```

## Input Arguments

### interface — Interface

data interface object | physical interface object | service interface object

Interface, specified as a `systemcomposer.interface.DataInterface`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### name — Element name

character vector | string

Element name, specified as a character vector or string. An element name must be a valid MATLAB variable name.

Data Types: char | string

### Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example:

```
addElement(interface,"newElement",DataType="double",Dimensions="2",Units="m/s",Complexity="complex",Minimum="0",Maximum="100",Description="Maintain altitude")
```

### DataType — Data type

character vector | string

Data type, specified as a character vector or string for a valid MATLAB data type. The default value is `double`.

Example: `addElement(interface,"newElement",DataType="double")`

Data Types: char | string

### Dimensions — Dimensions

character vector | string

Dimensions, specified as a character vector or string. The default value is `1`.

Example: `addElement(interface,"newElement",Dimensions="2")`

Data Types: `char` | `string`

**Units – Units**

character vector | string

Units, specified as a character vector or string.

Example: `addElement(interface,"newElement",Units="m/s")`

Data Types: `char` | `string`

**Complexity – Complexity**

character vector | string

Complexity, specified as a character vector or string. The default value is `real`. Other possible values are `complex` and `auto`.

Example: `addElement(interface,"newElement",Complexity="complex")`

Data Types: `char` | `string`

**Minimum – Minimum**

character vector | string

Minimum, specified as a character vector or string.

Example: `addElement(interface,"newElement",Minimum="0")`

Data Types: `char` | `string`

**Maximum – Maximum**

character vector | string

Maximum, specified as a character vector or string.

Example: `addElement(interface,"newElement",Maximum="100")`

Data Types: `char` | `string`

**Description – Description**

character vector | string

Description, specified as a character vector or string.

Example: `addElement(interface,"newElement",Description="Maintain altitude")`

Data Types: `char` | `string`

**Type – Physical domain**

character vector | string

Physical domain of physical element, specified as a character vector or string of a partial physical domain name. For a list of valid physical domain names, see “Domain-Specific Line Styles” (Simscape).

Example: `addElement(interface,"newElement",Type="electrical.six_phase")`

Data Types: `char` | `string`



## Output Arguments

### element — Element

data element object | physical element object | function element object

Element, returned as a `systemcomposer.interface.DataElement`, `systemcomposer.interface.PhysicalElement`, or `systemcomposer.interface.FunctionElement` object.

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>“Manage Interfaces with Data Dictionaries”</li> <li>“Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>“Create Architecture Model with Interfaces and Requirement Links”</li> <li>“Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>Pins or wires in a connector or harness.</li> <li>Messages transmitted across a bus.</li> <li>Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>“Create Interfaces”</li> <li>“Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

**See Also**

removeElement | getElement | getInterfaceNames | getInterface | setType |  
addInterface | addValueType | addPhysicalInterface | addServiceInterface

**Topics**

“Specify Physical Interfaces on Ports”

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## addElement

**Package:** `systemcomposer.view`

Add component to element group of view

### Syntax

```
addElement(elementGroup, component)
```

### Description

`addElement(elementGroup, component)` adds the component `component` to the element group `elementGroup` of an architecture view.

---

**Note** This function cannot be used when a selection query or grouping is defined on the view. To remove the query, run `removeQuery`.

---

### Examples

#### Add Elements to View

Open the keyless entry system example and create a view, `newView`.

```
scKeylessEntrySystem  
model = systemcomposer.loadModel("KeylessEntryArchitecture");  
view = model.createView("newView");
```

Open the Architecture Views Gallery to see `newView`.

```
model.openViews
```

Add an element to the view by path.

```
view.Root.addElement("KeylessEntryArchitecture/Lighting System/Headlights")
```

Add an element to the view by object.

```
component = model.lookup(Path="KeylessEntryArchitecture/Lighting System/Cabin Lights");  
view.Root.addElement(component)
```

### Input Arguments

#### **elementGroup** — Element group

element group object

Element group for view, specified as a `systemcomposer.view.ElementGroup` object.

**component — Component**

component object | variant component object | array of component objects | array of variant component objects | path to component | cell array of component paths

Component to remove from view, specified as a `systemcomposer.arch.Component` object, a `systemcomposer.arch.VariantComponent` object, an array of `systemcomposer.arch.Component` objects, an array of `systemcomposer.arch.VariantComponent` objects, the path to a component, or a cell array of component paths.

Example: "KeylessEntryArchitecture/Lighting System/Headlights"

Data Types: char | string

**More About****Definitions**

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>"Create Architecture Views Interactively"</li> <li>"Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

lookup | openViews | createView | getView | deleteView | systemcomposer.view.ElementGroup | systemcomposer.view.View | removeElement | getSubGroup | deleteSubGroup | createSubGroup | getQualifiedName

### Topics

“Create Architecture Views Interactively”  
 “Create Architectural Views Programmatically”

# addInterface

**Package:** systemcomposer.interface

Create named data interface in interface dictionary

## Syntax

```
interface = addInterface(dictionary,name)
interface = addInterface(dictionary,name,'SimulinkBus',busObject)
```

## Description

`interface = addInterface(dictionary,name)` adds the data interface specified by name `name` to the interface dictionary `dictionary`.

To remove an interface, use the `removeInterface` function.

`interface = addInterface(dictionary,name,'SimulinkBus',busObject)` constructs a data interface that mirrors an existing Simulink bus object.

## Examples

### Add Data Interface

Create a data dictionary, then add a data interface `newInterface`.

```
dictionary = systemcomposer.createDictionary("new_dictionary.sldd");
interface = addInterface(dictionary,"newInterface")
```

Create a new model and link the data dictionary. Then, open the **Interface Editor** to view the new interface.

```
arch = systemcomposer.createModel("newModel",true);
linkDictionary(arch,"new_dictionary.sldd");
```

### Add Simulink Bus Mirrored Data Interface

Create a dictionary, create a Simulink bus object, populate the bus object with two elements, and add the named data interface that mirrors the Simulink bus object to the dictionary.

```
dictionary = systemcomposer.createDictionary("new_dictionary.sldd");

busObj = Simulink.Bus;
elems(1) = Simulink.BusElement;
elems(1).Name = 'element_1';
elems(2) = Simulink.BusElement;
elems(2).Name = 'element_2';
busObj.Elements = elems;

interface = addInterface(dictionary,"newInterface",SimulinkBus=busObj);
```

Create a new model, link the data dictionary, and open the **Interface Editor**.

```
arch = systemcomposer.createModel("newModel",true);  
linkDictionary(arch,"new_dictionary.sldd");
```

## Input Arguments

### **dictionary** — Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

### **name** — Name of new data interface

character vector | string

Name of new data interface, specified as a character vector or string. This name must be a valid MATLAB identifier.

Example: "newInterface"

Data Types: `char` | `string`

### **busObject** — Simulink bus object that new data interface mirrors

bus object

Simulink bus object that new data interface mirrors, specified as a Simulink bus object.

## Output Arguments

### **interface** — New data interface

data interface object

New data interface, returned as a `systemcomposer.interface.DataInterface` object.



## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

`addElement` | `createDictionary` | `getInterface` | `getInterfaceNames` | `removeInterface` | `linkDictionary` | `Adapter` | `addPhysicalInterface` | `addValueType`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

# addPhysicalInterface

**Package:** systemcomposer.interface

Create named physical interface in interface dictionary

## Syntax

```
interface = addPhysicalInterface(dictionary,name)
```

## Description

`interface = addPhysicalInterface(dictionary,name)` adds the physical interface specified by the name `name` to the interface dictionary `dictionary`.

To remove an interface, use the `removeInterface` function.

## Examples

### Add Physical Interface

Create a data dictionary, then add a physical interface `newInterface`.

```
dictionary = systemcomposer.createDictionary("new_dictionary.slidd");
interface = addPhysicalInterface(dictionary,"newInterface")
```

Create a new model and link the data dictionary. Then, open the **Interface Editor** to view the new interface.

```
arch = systemcomposer.createModel("newModel",true);
linkDictionary(arch,"new_dictionary.slidd");
```

## Input Arguments

### **dictionary** — Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

### **name** — Name of new physical interface

character vector | string

Name of new physical interface, specified as a character vector or string. This name must be a valid MATLAB identifier.

Example: "newInterface"

Data Types: char | string

## Output Arguments

### interface — New physical interface

physical interface object

New physical interface, returned as a `systemcomposer.interface.PhysicalInterface` object.

## More About

### Definitions

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"

Term	Definition	Application	More Information
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	“Describe Component Behavior Using Simscape”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

**See Also**

[addElement](#) | [createDictionary](#) | [addInterface](#) | [getInterface](#) | [getInterfaceNames](#) | [removeInterface](#) | [linkDictionary](#) | [Adapter](#) | [addValueType](#)

**Topics**

[“Specify Physical Interfaces on Ports”](#)

[“Create Interfaces”](#)

[“Manage Interfaces with Data Dictionaries”](#)

## addFunction

**Package:** systemcomposer.arch

Add functions to architecture of software component

### Syntax

```
functions = addFunction(arch,functionNames)
```

### Description

`functions = addFunction(arch,functionNames)` adds a set of functions with the names specified, `functionNames` to the software architecture component `architecture`. The `addfunction` function adds functions to the software architecture of a component. Use `<component>.Architecture` to access the architecture of a component.

To remove a function, use the `destroy` function.

### Examples

#### Add Functions to Software Architecture Component

Create a model named `mySoftwareArchitecture` and get the root architecture.

```
model = systemcomposer.createModel("mySoftwareArchitecture","SoftwareArchitecture");
rootArch = model.Architecture
```

Architecture with properties:

```
        Name: 'mySoftwareArchitecture'
    Definition: Composition
        ...
    ExternalUID: ''
        Functions: []
```

Create a software component and two functions.

```
newComp = rootArch.addComponent("C1");
newFuncs = newComp.Architecture.addFunction({'f1','f2'});
rootArch
```

```
rootArch =
```

Architecture with properties:

```
        Name: 'mySoftwareArchitecture'
    Definition: Composition
        ...
    ExternalUID: ''
```



Functions: [1x2 systemcomposer.arch.Function]

## Input Arguments

### **arch — Architecture**

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **functionNames — Names of functions**

cell array of character vectors | array of strings

Names of functions, specified as a cell array of character vectors or an array of strings.

Data Types: `char` | `string`

## Output Arguments

### **functions — Handles to created functions**

array of function objects

Created functions, returned as an array of `systemcomposer.arch.Function` objects.

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Create Software Architecture from Component”</li> </ul>

Term	Definition	Application	More Information
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

## **Version History**

**Introduced in R2022a**

### **See Also**

`addComponent` | `systemcomposer.arch.Function`

### **Topics**

“Author and Extend Functions for Software Architectures”

# addParameter

**Package:** systemcomposer.arch

Add parameter to architecture

## Syntax

```
param = addParameter(arch,paramName)
param = addParameter(arch,Name,Value)
```

## Description

`param = addParameter(arch,paramName)` adds a parameter, `param`, with the name `paramName` to the architecture `arch`.

To delete a parameter, use the `destroy` function.

`param = addParameter(arch,Name,Value)` promotes a parameter from a component specified by a path to the parent architecture `arch`.

## Examples

### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the Pressure parameter on the RightWheel component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the Pressure parameter.

```
paramPressure.Type
```

```
ans =  
  ValueType with properties:  
  
      Name: 'Pressure'  
    DataType: 'double'  
  Dimensions: '[1 1]'  
      Units: 'psi'  
  Complexity: 'real'  
    Minimum: ''  
    Maximum: ''  
  Description: ''  
      Owner: [1x1 systemcomposer.arch.Architecture]  
      Model: [1x1 systemcomposer.arch.Model]  
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'  
  ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
          1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'31'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
          0
```

```
paramName =  
"Wear"
```

```

paramValue =
'0.25'

paramUnits =
'in'

isDefault = logical
    1

```

Get the LeftWheel component parameter values.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

paramValue =
'16'

paramUnits =
'in'

isDefault = logical
    1

```

```

paramName =
"Pressure"

paramValue =
'32'

paramUnits =
'psi'

isDefault = logical
    1

```

```

paramName =
"Wear"

paramValue =
'0.25'

paramUnits =
'in'

isDefault = logical
    1

```

First, check the evaluated RightWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)

```

```
        [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))
    end

    paramName =
    "Diameter"

    paramValue = 16

    paramUnits =
    'in'

    paramName =
    "Pressure"

    paramValue = 31

    paramUnits =
    'psi'

    paramName =
    "Wear"

    paramValue = 0.2500

    paramUnits =
    'in'
```

Check the evaluated LeftWheel parameters.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end

    paramName =
    "Diameter"

    paramValue = 16

    paramUnits =
    'in'

    paramName =
    "Pressure"

    paramValue = 32

    paramUnits =
    'psi'

    paramName =
    "Wear"

    paramValue = 0.2500

    paramUnits =
    'in'
```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.



```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

isDefault = logical
    1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure","34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'34'

paramUnits =
'psi'

isDefault = logical
    0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

isDefault = logical
    1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";
pressureParam

pressureParam =
    Parameter with properties:
```

```
Name: "LeftWheel.Pressure"  
Value: '30'  
Type: [1x1 systemcomposer.ValueType]  
Parent: [1x1 systemcomposer.arch.Architecture]  
Unit: 'psi'
```

Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =  
  Parameter with properties:  
  
  Name: 'Pressure'  
  Value: '30'  
  Type: [1x1 systemcomposer.ValueType]  
  Parent: [1x1 systemcomposer.arch.Component]  
  Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);  
pressureParam
```

```
pressureParam =  
  Parameter with properties:  
  
  Name: "LeftWheel.Pressure"  
  Value: '32'  
  Type: [1x1 systemcomposer.ValueType]  
  Parent: [1x1 systemcomposer.arch.Architecture]  
  Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");  
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;  
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce
```

```
noiseReduce =
  Parameter with properties:
    Name: "noiseReduction"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the Muffler component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");
save(model)
save(topModel)
```

## Input Arguments

### **arch — Architecture**

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **paramName — Parameter name**

character vector | string

Parameter name, specified as a character vector or string.

Example: "GainArg"

Data Types: char | string

### **Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `addParameter(arch,Path="Propeller/Hub",Parameters="all")`

### **Path — Path to component with parameter**

character vector | string

Path to component with parameter, specified as a character vector or string.

Example: `addParameter(arch,Path="Propeller/Hub")`

Data Types: char | string

### Parameters — Parameters to promote

"all" (default) | array of strings

Parameters to promote, specified as "all" or an array of strings.

Data Types: char | string

## Output Arguments

### param — Parameter

parameter object

Parameter, returned as a `systemcomposer.arch.Parameter` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022b

### See Also

getParameter | resetToDefault | getParameterPromotedFrom |  
getEvaluatedParameterValue | getParameterValue | setParameterValue | setUnit |  
getParameterNames | resetParameterToDefault

### Topics

“Author Parameters in System Composer Using Parameter Editor”  
“Access Model Arguments as Parameters on Reference Components”  
“Use Parameters to Store Instance Values with Components”

## addPort

**Package:** systemcomposer.arch

Add ports to architecture

### Syntax

```
ports = addPort(arch, portNames, portTypes)
ports = addPort(arch, portNames, portTypes, stereotypes)
```

### Description

`ports = addPort(arch, portNames, portTypes)` adds a set of ports with specified names `portNames` and types `portTypes`. The `addPort` function adds ports to the architecture of a component or the root architecture of the model. Use `<component>.Architecture` to access the architecture of a component.

To remove a port, use the `destroy` function.

`ports = addPort(arch, portNames, portTypes, stereotypes)` also applies stereotypes specified in `stereotypes` to a set of new ports.

### Examples

#### Add Port to Architecture

Create a model, get the root architecture, add a component, and add a port.

```
model = systemcomposer.createModel("archModel", true);
rootArch = get(model, "Architecture");
newComponent = addComponent(rootArch, "NewComponent");
newPort = addPort(newComponent.Architecture, "NewCompPort", "in")
```

`newPort =`

ArchitecturePort with properties:

```
    Parent: [1x1 systemcomposer.arch.Architecture]
      Name: 'NewCompPort'
    Direction: Input
  InterfaceName: ''
    Interface: [0x0 systemcomposer.interface.DataInterface]
   Connectors: [0x0 systemcomposer.arch.Connector]
    Connected: 0
      Model: [1x1 systemcomposer.arch.Model]
 SimulinkHandle: 57.0018
 SimulinkModelHandle: 10.0018
```



```

        UUID: 'f3dd03e1-af14-47ed-88c8-0ce301b2da5f'
    ExternalUID: ''

```

## Input Arguments

### **arch — Architecture**

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **portNames — Names of ports**

cell array of character vectors | array of strings | character vector | string

Names of ports, specified as a cell array of character vectors or an array of strings. If necessary, System Composer appends a number to the port name to ensure uniqueness. The size of `portNames`, `portTypes`, and `stereotypes` must be the same.

Data Types: char | string

### **portTypes — Port types**

cell array of character vectors | array of strings | character vector | string

Port types, specified as a cell array of character vectors or an array of strings. Available port types follow:

- "in"
- "out"
- "physical"
- "client" for software architectures
- "server" for software architectures

Data Types: char | string

### **stereotypes — Stereotypes to apply to ports**

array of stereotype objects

Stereotypes to apply to ports, specified as an array of `systemcomposer.profile.Stereotype` objects. Each stereotype in the array must either be a stereotype that applies to all element types or a port stereotype.

## Output Arguments

### **ports — Created ports**

array of ports

Created ports, returned as an array of `systemcomposer.arch.ArchitecturePort` objects.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"

Term	Definition	Application	More Information
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on Ports”
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	“Describe Component Behavior Using Simscape”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”

Term	Definition	Application	More Information
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

`destroy` | `systemcomposer.arch.BasePort` | `addComponent` | `connect` | `Component`

### Topics

“Ports”

# addProperty

**Package:** `systemcomposer.profile`

Define custom property for stereotype

## Syntax

```
property = addProperty(stereotype, name)
property = addProperty(stereotype, name, Name, Value)
```

## Description

`property = addProperty(stereotype, name)` returns a new property definition with name that is contained in `stereotype`.

To remove a property, use the `removeProperty` function.

`property = addProperty(stereotype, name, Name, Value)` returns a property definition that is configured with specified property values.

## Examples

### Add Property

Add a component stereotype and add a `VoltageRating` property with value 5.

```
profile = systemcomposer.profile.Profile.createProfile("myProfile");
stereotype = addStereotype(profile, "electricalComponent", AppliesTo="Component");
property = addProperty(stereotype, "VoltageRating", DefaultValue="5");
```

## Input Arguments

### stereotype — Stereotype

stereotype object

Stereotype, specified as a `systemcomposer.profile.Stereotype` object.

### name — Name of property

character vector | string

Name of property unique within the stereotype, specified as a character vector or string.

Data Types: `char` | `string`

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `addProperty(stereotype, "Amount", Type="double")`

### **Type — Property data type**

`double (default) | single | int64 | int32 | int16 | int8 | uint64 | uint32 | uint8 | boolean | string | enumeration class name`

Type of this property. One of valid data types or the name of a MATLAB class that defines an enumeration. For more information, see “Use Enumerated Data in Simulink Models”.

Example: `addProperty(stereotype, "Color", Type="BasicColors")`

Data Types: `char | string`

### **Dimensions — Dimensions of property**

positive integer array

Dimensions of property, specified as a positive integer array. Empty implies no restriction.

Example: `addProperty(stereotype, "Amount", Dimensions=2)`

Data Types: `double`

### **Min — Minimum value**

numeric

Optional minimum value of this property. To set both 'Min' and 'Max' together, use the `setMinAndMax` method.

Example: `setMinAndMax(property, min, max)`

Example: `addProperty(stereotype, "Amount", Min="0")`

Data Types: `double`

### **Max — Maximum value**

numeric

Optional maximum value of this property. To set both 'Min' and 'Max' together, use the `setMinAndMax` method.

Example: `setMinAndMax(property, min, max)`

Example: `addProperty(stereotype, "Amount", Max="100")`

Data Types: `double`

### **Units — Property units**

character vector | string

Units of the property value, specified as a character vector or string. If specified, all values of this property on model elements are checked for consistency with these units according to Simulink unit checking rules. For more information, see “Unit Consistency Checking and Propagation”.

Example: `addProperty(stereotype, "Amount", Units="kg")`

Data Types: `char | string`

### **DefaultValue — Default value**

character vector | string

Default value of this property, specified as a character vector or string that can be evaluated depending on the Type.

Data Types: `char` | `string`

## Output Arguments

### **property** — Created property

property object

Created property, returned as a `systemcomposer.profile.Property` object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>



Term	Definition	Application	More Information
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

getProperty | setProperty | removeProperty

### Topics

“Define Profiles and Stereotypes”

“Set Properties for Analysis”

## addReference

**Package:** `systemcomposer.interface`

Add reference to dictionary

### Syntax

```
addReference(dictionary, reference, collisionResolutionOption)
```

### Description

`addReference(dictionary, reference, collisionResolutionOption)` adds a referenced dictionary to a dictionary in a System Composer model.

### Examples

#### Add Referenced Dictionary

Add a data interface `newInterface` to the local interface dictionary of the model. Save the local interface dictionary to a shared dictionary as an SLDD file.

```
arch = systemcomposer.createModel("newModel", true);  
addInterface(arch.InterfaceDictionary, "newInterface");  
saveToDictionary(arch, "TopDictionary")  
topDictionary = systemcomposer.openDictionary("TopDictionary.sldd");
```

Create a new dictionary and add it as a reference to the existing dictionary.

```
refDictionary = systemcomposer.createDictionary("ReferenceDictionary.sldd");  
addReference(topDictionary, "ReferenceDictionary.sldd")
```

Confirm in the **Model Explorer**.

### Input Arguments

#### **dictionary** – Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

#### **reference** – Referenced dictionary

character vector | string

Referenced dictionary, specified as a character vector or string of the name of the referenced dictionary with the `.sldd` extension.

Example: `"ReferenceDictionary.sldd"`

Data Types: char | string

### collisionResolutionOption — Collision resolution option

"Unspecified" (default) | "KeepTop" | "KeepReference"

Collision resolution option if there is a conflict between two interfaces with the same name in the dictionaries, specified as one of the following:

- "KeepTop" to keep the interface from the top dictionary and remove the one in the reference dictionary.
- "KeepReference" to keep the interface from the reference dictionary and remove the one in the top dictionary.
- "Unspecified", which will error if any conflicts exist when creating the reference.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• "Create Architecture Model with Interfaces and Requirement Links"</li> <li>• "Define Port Interfaces Between Components"</li> </ul>

Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021a

### See Also

[saveToDictionary](#) | [createDictionary](#) | [openDictionary](#) | [linkDictionary](#) | [unlinkDictionary](#) | [removeReference](#)

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## addServiceInterface

**Package:** `systemcomposer.interface`

Create named service interface in interface dictionary

### Syntax

```
interface = addServiceInterface(dictionary,name)
```

### Description

`interface = addServiceInterface(dictionary,name)` adds the service interface specified by the name `name` to the interface dictionary `dictionary`.

To remove an interface, use the `removeInterface` function.

### Examples

#### Add Service Interface

Create a data dictionary, then add a service interface named `newInterface`.

```
dictionary = systemcomposer.createDictionary("new_dictionary.slidd");  
interface = addServiceInterface(dictionary,"newInterface")
```

Create a new model and link the data dictionary. Then, open the **Interface Editor** to view the new interface.

```
arch = systemcomposer.createModel("newModel",true);  
linkDictionary(arch,"new_dictionary.slidd");
```

### Input Arguments

#### **dictionary** — Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

#### **name** — Name of new service interface

character vector | string

Name of new service interface, specified as a character vector or string. This name must be a valid MATLAB identifier.

Example: `"newInterface"`

Data Types: char | string

## Output Arguments

### interface — New service interface

service interface object

New service interface, returned as a `systemcomposer.interface.ServiceInterface` object.

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”

Term	Definition	Application	More Information
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution — When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> <li>• Asynchronous execution — When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul>	<code>systemcomposer.interface.FunctionElement</code>
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	<code>systemcomposer.interface.FunctionArgument</code>



Term	Definition	Application	More Information
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	"Class Diagram View of Software Architectures"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• "Create Architecture Model with Interfaces and Requirement Links"</li> <li>• "Define Port Interfaces Between Components"</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• "Create Interfaces"</li> <li>• "Assign Interfaces to Ports"</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2022a

**See Also**

addElement | createDictionary | addInterface | getInterface | getInterfaceNames |  
removeInterface | linkDictionary | Adapter | addValueType | getFunctionArgument |  
setAsynchronous | setFunctionPrototype

**Topics**

“Author Service Interfaces for Client-Server Communication”

“Client-Server Interfaces in Class Diagram View”

“Define Port Interfaces Between Components”

## addStereotype

**Package:** systemcomposer.profile

Add stereotype to profile

### Syntax

```
stereotype = addStereotype(profile,name)
stereotype = addStereotype( ____,Name,Value)
```

### Description

`stereotype = addStereotype(profile,name)` adds a new stereotype with a specified name name to a profile profile.

`stereotype = addStereotype( ____,Name,Value)` adds a new stereotype with the previous input arguments and specifies properties for the stereotype.

### Examples

#### Add Component Stereotype

Add a component stereotype to a profile.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
stereotype = addStereotype(profile,"electricalComponent",AppliesTo="Component")
```

```
stereotype =
  Stereotype with properties:
      Name: 'electricalComponent'
  Description: ''
      Parent: [0x0 systemcomposer.profile.Stereotype]
  AppliesTo: 'Component'
      Abstract: 0
      Icon: ''
  ComponentHeaderColor: [210 210 210]
  ConnectorLineColor: [168 168 168]
  ConnectorLineStyle: 'Default'
  FullyQualifiedname: 'LatencyProfile.electricalComponent'
      Profile: [1x1 systemcomposer.profile.Profile]
  OwnedProperties: [0x0 systemcomposer.profile.Property]
  Properties: [0x0 systemcomposer.profile.Property]

close(profile,true)
```

## Input Arguments

### profile — Profile

profile object

Profile, specified as a `systemcomposer.profile.Profile` object.

### name — Stereotype name

character vector | string

Stereotype name, specified as a character vector or string. The name of the stereotype must be unique within the profile.

Data Types: char | string

### Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `addStereotype(profile, 'electricalComponent', AppliesTo="Component")`

### Description — Description text for stereotype

character vector | string

Description text for stereotype, specified as a character vector or string.

Example: `addStereotype(profile, 'electricalComponent', Description="These components are electrical")`

Data Types: char | string

### Icon — Icon name for stereotype

character vector | string

Icon name for stereotype, specified as a character vector or string. Built in options include:

- "default"
- "application"
- "channel"
- "controller"
- "database"
- "devicedriver"
- "memory"
- "network"
- "plant"
- "sensor"
- "subsystem"
- "transmitter"

This name-value argument is only valid for component stereotypes. The element a stereotype applies to is set with the `AppliesTo` name-value argument.

Example: `addStereotype(profile, "electricalComponent", Icon="default")`

Data Types: `char` | `string`

### **Parent — Stereotype from which stereotype inherits properties**

stereotype object

Stereotype from which stereotype inherits properties, specified as a `systemcomposer.profile.Stereotype` object.

Example: `addStereotype(profile, "electricalComponent", Parent=baseStereotype)`

### **AppliesTo — Element type to which stereotype can be applied**

`"` (default) | `"Component"` | `"Port"` | `"Connector"` | `"Interface"` | `"Function"` | `"Requirement"` | `"Link"`

Element type to which stereotype can be applied, specified as one of these options:

- `"` to apply stereotype to all element types
- `"Component"`
- `"Port"`
- `"Connector"`
- `"Interface"`
- `"Function"`, which is only available for software architectures
- `"Requirement"`, to be used with Requirements Toolbox
- `"Link"`, to be used with Requirements Toolbox

Example: `addStereotype(profile, "electricalComponent", AppliesTo="Port")`

Data Types: `char` | `string`

### **Abstract — Whether stereotype is abstract**

`false` or `0` (default) | `true` or `1`

Whether stereotype is abstract, specified as a logical. If `true`, then the stereotype cannot be directly applied on model elements, but instead serves as a parent for other stereotypes.

Example: `addStereotype(profile, 'electricalComponent', 'Abstract', true)`

Data Types: `logical`

### **ComponentHeaderColor — Component header color**

`1x3 uint32` row vector

Component header color, specified as a `1x3 uint32` row vector in the form `[Red Green Blue]`.

This name-value argument is only valid for component stereotypes. The element a stereotype applies to is set with the `AppliesTo` name-value argument.

Example: `addStereotype(profile, 'electricalComponent', 'ComponentHeaderColor', [206 232 246])`

Data Types: `uint32`

**ConnectorLineColor — Connector line color**

1x3 uint32 row vector

Connector line color, specified as a 1x3 uint32 row vector in the form [Red Green Blue].

This name-value argument is only valid for connector, port, and interface stereotypes. The element a stereotype applies to is set with the `AppliesTo` name-value argument.

Example: `addStereotype(profile, 'electricalComponent', 'ConnectorLineColor', [206 232 246])`

Data Types: uint32

**ConnectorLineStyle — Connector line style**

character vector | string

Connector line style name, specified as a character vector or string. Options include:

- "Default"
- "Dot"
- "Dash"
- "Dash Dot"
- "Dash Dot Dot"

This name-value argument is only valid for connector, port, and interface stereotypes. The element a stereotype applies to is set with the `AppliesTo` name-value argument.

Data Types: char | string

**Output Arguments****stereotype — Created stereotype**

stereotype object

Created stereotype, returned as a `systemcomposer.profile.Stereotype` object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

[getStereotype](#) | [getDefaultStereotype](#) | [setDefaultStereotype](#) | [removeStereotype](#)



**Topics**

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

## addValueType

**Package:** `systemcomposer.interface`

Create named value type in interface dictionary

### Syntax

```
valueType = addValueType(dictionary,name)
valueType = addValueType(dictionary,name,Name,Value)
```

### Description

`valueType = addValueType(dictionary,name)` adds a named value type to a specified interface dictionary.

To remove a value type, use the `destroy` function.

`valueType = addValueType(dictionary,name,Name,Value)` adds a named value type to a specified interface dictionary with additional options.

### Examples

#### Add Value Type

Create a data dictionary and add a value type `airSpeed`.

```
dictionary = systemcomposer.createDictionary("new_dictionary.slidd");
airSpeedType = addValueType(dictionary,"airSpeed")
```

Create a new model, link the data dictionary to the model, and view the Interface Editor to confirm the existence of the new value type `airSpeed`.

```
arch = systemcomposer.createModel("newModel",true);
linkDictionary(arch,"new_dictionary.slidd");
```

### Input Arguments

#### **dictionary** – Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

#### **name** – Name of new value type

character vector | string

Name of new value type, specified as a character vector or string. This name must be a valid MATLAB identifier.

Example: "airSpeed"

Data Types: char | string

### **Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example:

```
addValueType(dictionary,"airSpeed",DataType="double",Dimensions="2",Units="m/s",Complexity="complex",Minimum="0",Maximum="100",Description="Maintain altitude")
```

### **DataType — Data type of value type**

character vector | string

Data type of value type, specified as a character vector or string for a valid MATLAB data type. The default value is `double`.

Example: `addValueType(dictionary,"airSpeed",DataType="double")`

Data Types: char | string

### **Dimensions — Dimensions of value type**

character vector | string

Dimensions of value type, specified as a character vector or string. The default value is 1.

Example: `addValueType(dictionary,"airSpeed",Dimensions="2")`

Data Types: char | string

### **Units — Units of value type**

character vector | string

Units of value type, specified as a character vector or string.

Example: `addValueType(dictionary,"airSpeed",Units="m/s")`

Data Types: char | string

### **Complexity — Complexity of value type**

character vector | string

Complexity of value type, specified as a character vector or string. The default value is `real`. Other possible values are `complex` and `auto`.

Example: `addValueType(dictionary,"airSpeed",Complexity="complex")`

Data Types: char | string

### **Minimum — Minimum of value type**

character vector | string

Minimum of value type, specified as a character vector or string.

Example: `addValueType(dictionary,"airSpeed",Minimum="0")`

Data Types: `char | string`

**Maximum – Maximum of value type**

character vector | string

Maximum of value type, specified as a character vector or string.

Example: `addValueType(dictionary,"airSpeed",Maximum="100")`

Data Types: `char | string`

**Description – Description of value type**

character vector | string

Description of value type, specified as a character vector or string.

Example: `addValueType(dictionary,"airSpeed",Description="Maintain altitude")`

Data Types: `char | string`

**Output Arguments**

**valueType – Value type**

value type object

Value type, returned as a `systemcomposer.ValueType` object.

**More About**

**Definitions**

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>“Manage Interfaces with Data Dictionaries”</li> <li>“Reference Data Dictionaries”</li> </ul>

Term	Definition	Application	More Information
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`addElement` | `createDictionary` | `getInterface` | `getInterfaceNames` | `removeInterface` | `linkDictionary` | `Adapter` | `addPhysicalInterface` | `addInterface`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

# addVariantComponent

**Package:** systemcomposer.arch

Add variant components to architecture

## Syntax

```
variants = addVariantComponent(arch,variantComponents)
variants = addVariantComponent( ____, 'Position', position)
```

## Description

`variants = addVariantComponent(arch,variantComponents)` adds a set of variant components specified by the array of names.

To remove a variant component, use the `destroy` function.

`variants = addVariantComponent( ____, 'Position', position)` creates variant components in the architecture at a given position.

## Examples

### Create Variant Components

Create a model, get its root architecture, and create two variant components.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
names = ["Component1","Component2"];
variantComps = addVariantComponent(arch,names)
```

`variantComps=1x2 object`  
 1x2 VariantComponent array with properties:

```
Architecture
Name
Parent
Ports
OwnedPorts
OwnedArchitecture
Position
Model
SimulinkHandle
SimulinkModelHandle
UUID
ExternalUID
```

## Input Arguments

### **arch — Architecture**

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **variantComponents — Names of variant components**

cell array of character vectors | array of strings

Names of variant components, specified as a cell array of character vectors or an array of strings.

Data Types: `char` | `string`

### **position — Vector that specifies location of top corner and bottom corner of component**

1x4 numeric array

Vector that specifies location of top corner and bottom corner of component, specified as a 1x4 numeric array. The array denotes the top corner in terms of its x and y coordinates followed by the x and y coordinates of the bottom corner. When adding more than one variant component, a matrix of size [Nx4] may be specified where N is the number of variant components being added.

Data Types: `double`

## Output Arguments

### **variants — Variant components**

array of components

Variant components, returned as an array of `systemcomposer.arch.VariantComponent` objects. This array is the same size as `variantComponents`.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>



Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	"Create Architecture Model with Interfaces and Requirement Links"
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## **Version History**

**Introduced in R2019a**

### **See Also**

addPort | connect | addChoice | getActiveChoice | setActiveChoice | Variant Component

### **Topics**

"Create Variants"

# allocate

**Package:** systemcomposer.allocation

Create new allocation

## Syntax

```
allocation = allocate(allocScenario,sourceElement,targetElement)
```

## Description

`allocation = allocate(allocScenario,sourceElement,targetElement)` creates a new allocation between the source element `sourceElement` and target element `targetElement`.

To remove an allocation, use the `deallocate` function.

## Examples

### Create Allocation Set and Allocate Elements Between Models

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario,sourceComp,targetComp);
```

Save the allocation set.

```
save(allocSet)
```

Open the **Allocation Editor**.

```
systemcomposer.allocation.editor
```

## Input Arguments

### **allocScenario** — Allocation scenario

allocation scenario object

Allocation scenario , specified as a `systemcomposer.allocation.AllocationScenario` object.

**sourceElement — Source element**

element object

Source element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

**targetElement — Target element**

element object

Target element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

**Output Arguments****allocation — Allocation**

allocation object

Allocation between source and target element, returned as a `systemcomposer.allocation.Allocation` object.

**More About****Definitions**

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called Scenario 1.	“Systems Engineering Approach for SoC Applications”

Term	Definition	Application	More Information
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>• “Establish Traceability Between Architectures and Requirements”</li> <li>• “Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

getAllocation | getAllocatedFrom | getAllocatedTo | deallocate | destroy | getScenario | createAllocationSet

### Topics

“Create and Manage Allocations Programmatically”

## AnyComponent

**Package:** `systemcomposer.query`

Create query to select all components in model

### Syntax

```
query = AnyComponent
```

### Description

`query = AnyComponent` creates a query `query` that the `find` and `createView` functions use to select all components in the model.

### Examples

#### Select All Components in Model

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
scKeylessEntrySystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query to find all components and list the second component.

```
constraint = AnyComponent;  
components = find(model,constraint,Recurse=true,IncludeReferenceModels=true);  
comp = components(2)
```

```
comp = 1x1 cell array  
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Actuator'}
```

### Output Arguments

#### **query** — Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `getQualifiedName`

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”



# applyProfile

**Package:** systemcomposer.arch

Apply profile to model

## Syntax

```
applyProfile(model,profileFile)
applyProfile(model,profileFile,Name,Value)
```

## Description

applyProfile(model,profileFile) applies a profile to an architecture model and makes all the constituent stereotypes available.

applyProfile(model,profileFile,Name,Value) specifies additional options using one or more name-value arguments.

## Examples

### Apply Profile

Create a model.

```
model = systemcomposer.createModel("archModel",true);
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

## Input Arguments

### model — Architecture model

model object

Architecture model, specified as a systemcomposer.arch.Model object.

### profileFile — Name of profile

character vector | string

Name of profile, specified as a character vector or string.

Example: "SystemProfile"

Data Types: char | string

## Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `applyProfile(model,profileFile,IncludeReferenceModels=true)`

## IncludeReferenceModels — Option to apply profile to all referenced component models

false or 0 (default) | true or 1

Option to apply profile to all referenced component models, specified as a logical.

Example: `applyProfile(model,profileFile,IncludeReferenceModels=true)`

Data Types: logical

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

`removeProfile` | `createProfile`

### Topics

“Define Profiles and Stereotypes”

# applyStereotype

**Package:** systemcomposer.arch

Apply stereotype to architecture model element

## Syntax

```
applyStereotype(element, stereotype)
```

## Description

`applyStereotype(element, stereotype)` applies a stereotype to an architecture model element if the stereotype is not already applied to a model element. Stereotypes can be applied to architecture, component, port, connector, interface, and function model elements. The function model element is only available in software architectures.

## Examples

### Apply Stereotype

Create a model with a component.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
comp = addComponent(arch, "Component");
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

Apply the stereotype to the component and get the stereotypes on the component.

```
comp.applyStereotype("LatencyProfile.LatencyBase");
stereotypes = getStereotypes(comp)
```

```
stereotypes =
```

```
    1x1 cell array
```

```
{'LatencyProfile.LatencyBase'}
```

## Input Arguments

### element — Architectural element

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### stereotype — Name of stereotype

character vector | string

Name of stereotype, specified as a character vector or string in the form "`<profile>.<stereotype>`". The profile must already be applied to the model.

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”



Term	Definition	Application	More Information
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2019a

### See Also

[batchApplyStereotype](#) | [removeStereotype](#) | [getStereotypes](#) | [getStereotypeProperties](#)

### Topics

"Use Stereotypes and Profiles"

## batchApplyStereotype

**Package:** systemcomposer.arch

Apply stereotype to all elements in architecture

### Syntax

```
batchApplyStereotype(arch, elementType, stereotype)
batchApplyStereotype( ____, 'Recurse', flag)
```

### Description

`batchApplyStereotype(arch, elementType, stereotype)` applies the stereotype `stereotype` to all elements that match the element type `elementType` within the architecture `arch`.

`batchApplyStereotype( ____, 'Recurse', flag)` applies the stereotype `stereotype` to all elements that match the element type `elementType` within the architecture `arch` and recursively to its sub-architectures according to the value of `flag`.

### Examples

#### Apply Stereotype to All Connectors

Create a profile, add a connector stereotype, and add a property with a default value. Open the Profile Editor to inspect the profile.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
stereotype = addStereotype(profile, "standardConn", AppliesTo="Connector");
stereotype.addProperty("latency", Type="double", DefaultValue="10");
systemcomposer.profile.editor(profile)
```

Create a model with three components, ports, and connectors between them. Improve the model layout.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName, true);
rootArch = get(arch, "Architecture");
names = ["Component1", "Component2", "Component3"];
newComponents = addComponent(rootArch, names);
outPort1 = addPort(newComponents(1).Architecture, "testSig1", "out");
inPort1 = addPort(newComponents(2).Architecture, "testSig1", "in");
outPort2 = addPort(newComponents(2).Architecture, "testSig2", "out");
inPort2 = addPort(newComponents(3).Architecture, "testSig2", "in");
conn1 = connect(newComponents(1), newComponents(2));
conn2 = connect(newComponents(2), newComponents(3));
Simulink.BlockDiagram.arrangeSystem(modelName)
```

Apply the profile to the model.

```
arch.applyProfile("LatencyProfile");
```

Apply the connector stereotype to all the connectors in the architecture `rootArch`. Inspect the connectors in the **Property Inspector** to confirm the applied stereotypes.

```
batchApplyStereotype(rootArch, "Connector", "LatencyProfile.standardConn")
```

## Input Arguments

### arch — Architecture

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### elementType — Element type

"Component" | "Port" | "Connector" | "Interface" | "Function"

Element type, specified as "Component", "Port", "Connector", "Interface", or "Function". The element type "Function" is only available for software architectures.

Data Types: char | string

### stereotype — Stereotype to apply

character vector | string

Stereotype to apply, specified as a character vector or string in the form "<profile>.<stereotype>". This stereotype must be applicable for the element type.

Data Types: char | string

### flag — Whether to apply stereotype recursively

false or 0 (default) | true or 1

Whether to apply stereotype recursively, specified as a logical. If `flag` is 1 (true), the stereotype is applied to the elements in the architecture and its sub-architectures.

Data Types: logical

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

[removeStereotype](#) | [applyStereotype](#) | [getStereotypes](#)

### Topics

“Use Stereotypes and Profiles”

## close

**Package:** systemcomposer.profile

Close profile

### Syntax

```
close(profile, force)
```

### Description

`close(profile, force)` closes the profile and deletes it from the workspace. If there are any unsaved changes, you will receive an error unless the argument `force` is set to `true`.

---

**Tip** Use `closeAll` to force close all loaded profiles.

---

## Examples

### Close Profile

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency", ...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure", Type="boolean");
connLatency.addProperty("linkDistance", Type="double");

nodeLatency = profile.addStereotype("NodeLatency", ...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");

portLatency = profile.addStereotype("PortLatency", ...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth", Type="double");
portLatency.addProperty("dummy", Type="int32");

profile.save
```

Force close profile and attempt to inspect it.

```
profile.close(true)
profile
```

```
profile =
    handle to deleted Profile
```

## Input Arguments

### profile — Profile

profile object

Profile, specified as a `systemcomposer.profile.Profile` object.

### force — Whether to force close profile

false or 0 (default) | true or 1

Whether to force close profile, specified as a logical 1 (`true`) to close the profile without saving or 0 (`false`) to be prompted to save the profile before closing.

Data Types: `logical`

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”

Term	Definition	Application	More Information
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

`systemcomposer.profile.Profile | open | editor | load | find | closeAll | save`

### Topics

“Define Profiles and Stereotypes”



# close

**Package:** systemcomposer.arch

Close architecture model

## Syntax

```
close(model)
```

## Description

`close(model)` closes the specified model in System Composer.

## Examples

### Create, Open, and Close Model

```
model = systemcomposer.createModel("modelName");
open(model)
close(model)
```

## Input Arguments

### model — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## **Version History**

**Introduced in R2019a**

### **See Also**

`createModel` | `save` | `loadModel`

### **Topics**

“Create Architecture Model”

## close

**Package:** systemcomposer.allocation

Close allocation set

### Syntax

```
close(allocSet, force)
```

### Description

`close(allocSet, force)` closes the allocation set `allocSet`. If there are any unsaved changes, you will receive an error unless the argument `force` is `true`.

---

**Tip** Use `closeAll` to close all loaded allocation sets.

---

## Examples

### Close Allocation Set Without Saving

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation", true);
sourceComp = addComponent(get(mSource, "Architecture"), "Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation", true);
targetComp = addComponent(get(mTarget, "Architecture"), "Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation", ...
    "Source_Model_Allocation", "Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet, "Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario, sourceComp, targetComp);
```

Close the allocation set without saving.

```
allocSet.close(true)
```

Open the **Allocation Editor**.

```
systemcomposer.allocation.editor
```

## Input Arguments

**allocSet** — Allocation set

allocation set object

Allocation set, specified as a `systemcomposer.allocation.AllocationSet` object.

**force — Force close**

false or 0 (default) | true or 1

Force close allocation set, specified as a logical.

Data Types: `logical`

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

`createScenario` | `deleteScenario` | `getScenario` | `load` | `closeAll` | `synchronizeChanges`

### Topics

“Create and Manage Allocations Programmatically”

## systemcomposer.allocation.AllocationSet.closeAll

Close all open allocation sets

### Syntax

```
systemcomposer.allocation.AllocationSet.closeAll
```

### Description

`systemcomposer.allocation.AllocationSet.closeAll` closes all allocation sets without saving.

---

**Tip** Use `close` to close one allocation set.

---

### Examples

#### Close All Allocation Sets Without Saving

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario,sourceComp,targetComp);
```

Close all allocation sets without saving.

```
systemcomposer.allocation.AllocationSet.closeAll
```

Open the **Allocation Editor**.

systemcomposer.allocation.editor

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

createScenario | deleteScenario | getScenario | load | close | synchronizeChanges | find

### Topics

“Create and Manage Allocations Programmatically”

## systemcomposer.profile.Profile.closeAll

Close all open profiles

### Syntax

```
systemcomposer.profile.Profile.closeAll
```

### Description

`systemcomposer.profile.Profile.closeAll` force closes all open profiles without saving and deletes them from the workspace.

---

**Tip** Use `close` to close one open profile.

---

### Examples

#### Close All Profiles

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Close all open profiles and attempt to inspect one.

```
systemcomposer.profile.Profile.closeAll
profile
```



profile =

handle to deleted Profile

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

**See Also**

systemcomposer.profile.Profile | load | editor | open | find | close | save

**Topics**

“Define Profiles and Stereotypes”

# connect

**Package:** systemcomposer.arch

Create architecture model connections

## Syntax

```
connectors = connect(srcComponent, destComponent)
connectors = connect(arch, [srcComponent, srcComponent, ...], [destComponent,
destComponent, ...])
connectors = connect(arch, [], destComponent)
connectors = connect(arch, srcComponent, [])
connectors = connect(srcPort, destPort)
connectors = connect(srcPort, destPort, stereotype)
connectors = connect( ____, Name, Value)
```

## Description

`connectors = connect(srcComponent, destComponent)` connects the unconnected output ports of the source component `srcComponent` to the unconnected input ports of the destination component `destComponent` based on matching port names, and returns a handle to the connector. For physical connections, the connectors are nondirectional so the source and destination components can be interchanged.

To remove a connector, use the `destroy` function.

`connectors = connect(arch, [srcComponent, srcComponent, ...], [destComponent, destComponent, ...])` connects arrays of components in the architecture.

`connectors = connect(arch, [], destComponent)` connects a parent architecture input port to a destination child component.

`connectors = connect(arch, srcComponent, [])` connects a source child component to a parent architecture output port.

`connectors = connect(srcPort, destPort)` connects a source port and a destination port, or connects two nondirectional physical ports.

`connectors = connect(srcPort, destPort, stereotype)` connects two ports and applies a stereotype to the connector.

`connectors = connect( ____, Name, Value)` specifies options using one or more name-value arguments in addition to the input arguments in previous syntaxes.

## Examples

### Connect System Composer Components

Create and connect two components.

Create a top-level architecture model.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName,true);  
rootArch = get(arch,"Architecture");
```

Create two new components.

```
names = ["Component1","Component2"];  
newComponents = addComponent(rootArch,names);
```

Add ports to the components.

```
outPort1 = addPort(newComponents(1).Architecture,"testSig","out");  
inPort1 = addPort(newComponents(2).Architecture,"testSig","in");
```

Connect the components.

```
conns = connect(newComponents(1),newComponents(2));
```

Improve the model layout.

```
Simulink.BlockDiagram.arrangeSystem(modelName)
```

### **Connect System Composer Ports**

Create and connect two ports.

Create a top-level architecture model.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName,true);  
rootArch = get(arch,"Architecture");
```

Create two new components.

```
names = ["Component1","Component2"];  
newComponents = addComponent(rootArch,names);
```

Add ports to the components.

```
outPort1 = addPort(newComponents(1).Architecture,"testSig","out");  
inPort1 = addPort(newComponents(2).Architecture,"testSig","in");
```

Extract the component ports.

```
srcPort = getPort(newComponents(1),"testSig");  
destPort = getPort(newComponents(2),"testSig");
```

Connect the ports.

```
conns = connect(srcPort,destPort);
```

Improve the model layout.

```
Simulink.BlockDiagram.arrangeSystem(modelName)
```

## Connect by Selecting Destination Element

Create and connect a destination architecture port interface element to a component.

Create a top-level architecture model.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName,true);
rootArch = get(arch,"Architecture");
```

Create a new component.

```
newComponent = addComponent(rootArch,"Component1");
```

Add destination architecture ports to the component and the architecture.

```
outPortComp = addPort(newComponent.Architecture,"testSig","out");
outPortArch = addPort(rootArch,"testSig","out");
```

Extract corresponding port objects.

```
compSrcPort = getPort(newComponent,"testSig");
archDestPort = getPort(rootArch,"testSig");
```

Add an interface and an interface element, and associate the interface with the architecture port.

```
interface = arch.InterfaceDictionary.addInterface("interface");
interface.addElement("x");
archDestPort.setInterface(interface);
```

Select an element on the architecture port and establish a connection.

```
conns = connect(compSrcPort,archDestPort,DestinationElement="x");
```

Improve the model layout.

```
Simulink.BlockDiagram.arrangeSystem(modelName)
```

## Input Arguments

### **arch** — Architecture

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **srcComponent** — Source component

component object | variant component object

Source component, specified as a `systemcomposer.arch.Component` or `systemcomposer.arch.VariantComponent` object.

### **destComponent** — Destination component

component object | variant component object

Destination component, specified as a `systemcomposer.arch.Component` or `systemcomposer.arch.VariantComponent` object.

**srcPort — Source port**

port object

Source port to connect, specified as a `systemcomposer.arch.ComponentPort` or `systemcomposer.arch.ArchitecturePort` object.

**destPort — Destination port**

port object

Destination port to connect, specified as a `systemcomposer.arch.ComponentPort` or `systemcomposer.arch.ArchitecturePort` object.

**stereotype — Stereotype**

character vector | string

Stereotype to apply to the connection, specified in the form "`<profile>.<stereotype>`".

Data Types: char | string

**Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `connect(archPort, compPort, SourceElement="a")`

**Stereotype — Option to apply stereotype to connector**

character vector | string

Option to apply stereotype to connector, specified in the form "`<profile>.<stereotype>`".

This name-value argument applies only when you connect components.

Example: `conns = connect(srcComp, destComp, Stereotype="GeneralProfile.ConnStereotype")`

Data Types: char | string

**Rule — Option to specify rule for connections**`"name" (default) | "interface"`

Option to specify rule for connections, specified as either `"name"` based on the name of ports or `"interface"` based on the interface name on ports.

This name-value argument applies only when you connect components.

Example: `conns = connect([srcComp1, srcComp2], [destComp1, destComp2], Rule="interface")`

Data Types: char | string

**MultipleOutputConnectors — Option to allow multiple destination components**`false or 0 (default) | true or 1`

Option to allow multiple destination components for the same source component, specified as a logical.

This name-value argument applies only when you connect components.

```
Example: conns = connect(srcComp,
[destComp1,destComp2],MultipleOutputConnectors=true)
```

Data Types: `logical`

### **SourceElement — Option to select source element for connection**

character vector | string

Option to select source element for connection, specified as a character vector or string of the name of the data element.

This name-value argument applies only when you connect ports.

```
Example: conns = connect(archSrcPort,compDestPort,SourceElement="x")
```

Data Types: `char` | `string`

### **DestinationElement — Option to select destination element for connection**

character vector | string

Option to select destination element for connection, specified as a character vector or string of the name of the data element.

This name-value argument applies only when you connect ports.

```
Example: conns = connect(compSrcPort,archDestPort,DestinationElement="x")
```

Data Types: `char` | `string`

### **Routing — Option to specify type of automatic line routing**

"smart" (default) | "on" | "off"

Option to specify type of automatic line routing, specified as one of the following:

- "smart" — Use automatic line routing that takes the best advantage of the blank spaces on the canvas and avoids overlapping other lines and labels.
- "on" — Use automatic line routing.
- "off" — Use no automatic line routing.

```
Example: conns = connect(srcPort,destPort,Routing="on")
```

Data Types: `char` | `string`

## **Output Arguments**

### **connectors — Created connections**

array of connections

Created connections, returned as an array of `systemcomposer.arch.Connector` or `systemcomposer.arch.PhysicalConnector` objects.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”



Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"

Term	Definition	Application	More Information
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2019a

### See Also

`openModel` | `createModel` | `addPort` | `getPort` | `addComponent` | `addElement` | `addInterface` | `setInterface` | `getSourceElement` | `getDestinationElement` | `Component`

### Topics

"Connections"

"Build Architecture Models Programmatically"

# systemcomposer.allocation.createAllocationSet

Create new allocation set

## Syntax

```
allocSet = systemcomposer.allocation.createAllocationSet(name, sourceModel, targetModel)
```

## Description

`allocSet = systemcomposer.allocation.createAllocationSet(name, sourceModel, targetModel)` creates a new allocation set with the given name in which the source and target models are provided.

## Examples

### Create Allocation Set and Open in Allocation Editor

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation", true);
sourceComp = addComponent(get(mSource, "Architecture"), "Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation", true);
targetComp = addComponent(get(mTarget, "Architecture"), "Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation", ...
    "Source_Model_Allocation", "Target_Model_Allocation");
```

Save the allocation set.

```
save(allocSet)
```

Open the **Allocation Editor**.

```
systemcomposer.allocation.editor
```

## Input Arguments

### **name** — Name of allocation set

character vector | string

Name of allocation set, specified as a character vector or string.

Example: "MyNewAllocation"

Data Types: char | string

### **sourceModel** — Source model for allocation

model object | character vector | string

Source model for allocation, specified as a `systemcomposer.arch.Model` object or the name of a model as a character vector or string.

Data Types: `char` | `string`

### **targetModel — Target model for allocation**

model object | character vector | string

Target model for allocation, specified as a `systemcomposer.arch.Model` object or the name of a model as a character vector or string.

Data Types: `char` | `string`

## **Output Arguments**

### **allocSet — Allocation set**

allocation set object

Allocation set created, returned as a `systemcomposer.allocation.AllocationSet` object.

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## **Version History**

**Introduced in R2020b**

### **See Also**

`load` | `open` | `closeAll`

**Topics**

“Create and Manage Allocations Programmatically”

## createAnonymousInterface

**Package:** `systemcomposer.arch`

(To be removed) Create and set anonymous interface for port

---

**Note** The `createAnonymousInterface` function is not recommended in R2021b. It has been replaced with the `createInterface` function. For further details, see “Compatibility Considerations”.

---

### Syntax

```
interface = createAnonymousInterface(port)
```

### Description

`interface = createAnonymousInterface(port)` creates and sets an anonymous interface for the specified port `port`.

### Input Arguments

**port — Port**

port object

Port, specified as a `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` object.

### Output Arguments

**interface — Data interface**

data interface object

Data interface, returned as a `systemcomposer.interface.DataInterface` object.

## Version History

**Introduced in R2019a**

**R2021b: `createAnonymousInterface` function is not recommended**

The `createAnonymousInterface` function is not recommended in R2021b. Use `createInterface` instead.

### See Also

`Component` | `createInterface` | `addValueType` | `systemcomposer.ValueType` | `addInterface` | `removeInterface`

**Topics**

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## createArchitectureModel

**Package:** systemcomposer.arch

Create architecture model from component

### Syntax

```
createArchitectureModel(component,modelName)
createArchitectureModel(component,modelName,modelType)
```

### Description

`createArchitectureModel(component,modelName)` creates an architecture model from the component `component` that references the model `modelName`.

---

**Note** Components with physical ports cannot be saved as architecture models, model references, software architectures, or Stateflow chart behaviors. Components with physical ports can only be saved as subsystem references or subsystem component behaviors.

---

`createArchitectureModel(component,modelName,modelType)` creates an architecture model of type `modelType` from the component `component` that references the model `modelName`.

### Examples

#### Create Architecture Model from Component

Save the component `robotComp` in the `Robot.slx` model and reference the model.

Create a model named `archModel.slx`.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
```

Add two components named "electricComp" and "robotComp" to the model.

```
names = ["electricComp","robotComp"];
comp = addComponent(arch,names);
```

Save the `robotComp` component in an architecture model so the component references the model `Robot.slx`.

```
createArchitectureModel(comp(2),"Robot");
```

#### Create Software Architecture Model from Component

Save the component `electricComp` in the `RobotSoftware.slx` model and reference the model.



Create a model named `archModel.slx`.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
```

Add two components named 'electricComp' and 'robotComp' to the model.

```
names = ["electricComp", "robotComp"];
comp = addComponent(arch, names);
```

Save the `electricComp` component in a software architecture model so the component references the model `RobotSoftware.slx`.

```
createArchitectureModel(comp(1), "RobotSoftware", "SoftwareArchitecture");
```

### Create AUTOSAR Architecture Model from Component

Save the component `throttlePositionControl` in the `autosarTpcSys.slx` model and reference the model.

Create a model named `archModel.slx`.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
```

Add two components named 'throttlePositionControl' and 'Sensor' to the model.

```
names = ["throttlePositionControl", "Sensor"];
comp = addComponent(arch, names);
```

Save the `throttlePositionControl` component in a software architecture model so the component references the model `autosarTpcSys.slx`.

```
createArchitectureModel(comp(1), "autosarTpcSys", "ClassicAUTOSARArchitecture");
```

## Input Arguments

### **component** — Component

component object

Component, specified as a `systemcomposer.arch.Component` object. The component must have an architecture with definition type `composition`. For other definition types, this function gives an error.

### **modelName** — Name of model

character vector | string

Name of model, specified as a character vector or string.

Example: "exMobileRobot"

Data Types: char | string

### **modelType** — Type of model

"Architecture" (default) | "SoftwareArchitecture" | "ClassicAUTOSARArchitecture" | "AdaptiveAUTOSARArchitecture"

Type of model, specified as one of these values:

- "Architecture" - An architecture model
- "SoftwareArchitecture" - A software architecture model
- "ClassicAUTOSARArchitecture" - A Classic AUTOSAR architecture model
- "AdaptiveAUTOSARArchitecture" - An Adaptive AUTOSAR architecture model

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• "Compose Architectures Visually"</li> <li>• "Author Parameters in System Composer Using Parameter Editor"</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	"Create Architecture Model with Interfaces and Requirement Links"

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>• “Author Software Architectures”</li> <li>• “Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

## Version History

Introduced in R2021b

### See Also

[inlineComponent](#) | [createSimulinkBehavior](#) | [createStateflowChartBehavior](#) | [extractArchitectureFromSimulink](#) | [linkToModel](#) | [isReference](#) | [Reference Component](#)

### Topics

["Implement Component Behavior Using Simulink"](#)

["Decompose and Reuse Components"](#)

["Implement Component Behavior Using Stateflow Charts"](#)

["Create Simulink Subsystem Behavior Using Subsystem Component"](#)

["Simulate and Deploy Software Architectures"](#)

## systemcomposer.createDictionary

Create data dictionary

### Syntax

```
dictionary = systemcomposer.createDictionary(dictionaryName)
```

### Description

`dictionary = systemcomposer.createDictionary(dictionaryName)` creates a new Simulink data dictionary to hold interfaces and returns the `systemcomposer.interface.Dictionary` object.

### Examples

#### Create New Dictionary

```
dictionary = systemcomposer.createDictionary("new_dictionary.sldd")
```

### Input Arguments

#### dictionaryName — Name of new data dictionary

character vector | string

Name of new data dictionary, specified as a character vector or string. The name must include the `.sldd` extension and must be a valid MATLAB identifier.

Example: "new\_dictionary.sldd"

Data Types: char | string

### Output Arguments

#### dictionary — Dictionary

dictionary object

Dictionary, returned as a `systemcomposer.interface.Dictionary` object.



## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>

Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

[addValueType](#) | [addInterface](#) | [linkDictionary](#) | [saveToDictionary](#) | [unlinkDictionary](#) | [openDictionary](#) | [addReference](#) | [removeReference](#)

### Topics

“Define Port Interfaces Between Components”

“Manage Interfaces with Data Dictionaries”

# createInterface

**Package:** systemcomposer.arch

Create and set owned interface for port

## Syntax

```
interface = createInterface(port,kind)
```

## Description

`interface = createInterface(port,kind)` creates and sets an owned interface for a port.

## Examples

### Create Owned Interface as Value Type

Create an architecture model `archModel`. Get the root architecture, then add a new component `newComponent` and a new port `newCompPort`. Create an owned interface for the port as a `ValueType`.

```
model = systemcomposer.createModel("archModel",true);
rootArch = get(model,"Architecture");
newComponent = addComponent(rootArch,"newComponent");
newPort = addPort(newComponent.Architecture,"newCompPort","in");
interface = newPort.createInterface("ValueType")
```

```
interface =
```

```
ValueType with properties:
```

```
    Name: ''
    DataType: 'double'
    Dimensions: '1'
    Units: ''
    Complexity: 'real'
    Minimum: '[]'
    Maximum: '[]'
    Description: ''
    Owner: [1x1 systemcomposer.arch.ArchitecturePort]
    Model: [1x1 systemcomposer.arch.Model]
    UUID: 'd23669e1-f26c-4c64-a482-a27a33ac6541'
    ExternalUUID: ''
```

### Create Owned Interface as Data Interface and Remove Owned Interface

Create an architecture model `archModel`. Get the root architecture, then add a new component `newComponent` and a new port `newCompPort`. Create an owned interface for the port as a `DataInterface`.

```

model = systemcomposer.createModel("archModel",true);
rootArch = get(model,"Architecture");
newComponent = addComponent(rootArch,"newComponent");
newPort = addPort(newComponent.Architecture,"newCompPort","in");
interface = newPort.createInterface("DataInterface");

```

Remove the owned interface from the port.

```
newPort.setInterface("");
```

### Create Owned Interface for Physical Port as Physical Domain

Create an architecture model `archModel`. Get the root architecture, then add a new component `newComponent` and a new physical port `newCompPort`. Create an owned interface for the physical port and set the physical domain `Domain` property.

```

model = systemcomposer.createModel("archModel",true);
rootArch = get(model,"Architecture");
newComponent = addComponent(rootArch,"newComponent");
newPort = addPort(newComponent.Architecture,"newCompPort","physical");
port = newComponent.getPort("newCompPort");
interface = port.createInterface("PhysicalDomain");
interface.Domain = "rotational.rotational"

```

```
interface =
```

```
PhysicalDomain with properties:
```

```

    Domain: 'foundation.mechanical.rotational.rotational'
    Owner: [1x1 systemcomposer.arch.ArchitecturePort]
    Model: [1x1 systemcomposer.arch.Model]
    UUID: '65f143cb-ed3a-49e1-bbc9-de89e84aa8e6'
    ExternalUID: ''

```

## Input Arguments

### port — Port

port object

Port, specified as a `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` object.

### kind — Kind of interface

"DataInterface" | "ValueType" | "PhysicalDomain"

Kind of interface, specified as one of these options:

- "DataInterface"
- "ValueType"
- "PhysicalDomain"

Data Types: char | string

## Output Arguments

### interface — Interface

data interface object | value type object | physical domain object

Interface, returned as a `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, or `systemcomposer.interface.PhysicalDomain` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>



Term	Definition	Application	More Information
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	“Define Physical Ports on Component”

Term	Definition	Application	More Information
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2021b

### See Also

`addValueType` | `createModel` | `addInterface` | `setType` | `createOwnedType` | `addPhysicalInterface` | `removeInterface`

### Topics

"Specify Physical Interfaces on Ports"  
 "Create Interfaces"  
 "Manage Interfaces with Data Dictionaries"

## createOwnedType

**Package:** systemcomposer.interface

Create owned value type on data element or function argument

### Syntax

```
ownedType = createOwnedType(dataElement)
ownedType = createOwnedType(dataElement, Name, Value)
```

### Description

`ownedType = createOwnedType(dataElement)` creates an owned value type on a data element or function argument.

An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.

`ownedType = createOwnedType(dataElement, Name, Value)` creates an owned value type on a data element or function argument with additional options.

### Examples

#### Create Owned Value Type on Data Element on Architecture Port

```
model = systemcomposer.createModel("archModel", true);
```

```
port = model.Architecture.addPort("inPort", "in");
interface = port.createInterface("DataInterface");
element = interface.addElement("newElement");
subInterface = element.createOwnedType
```

```
subInterface =
```

```
    ValueType with properties:
```

```
        Name: ''
        DataType: 'double'
        Dimensions: '1'
        Units: ''
        Complexity: 'real'
        Minimum: '[]'
        Maximum: '[]'
        Description: ''
        Owner: [1x1 systemcomposer.interface.DataElement]
        Model: [1x1 systemcomposer.arch.Model]
        UUID: 'd184ab90-2be9-4acc-9d94-ed62d0cf2827'
        ExternalUID: ''
```

Select the architecture port `inPort` on the architecture model and open the **Property Inspector** from the **Modeling** menu. Under **Open in Interface Editor**, select the edit link. In the **Interface**

**Editor**, enter the Port Interface View. Observe the new data element newElement under the port inPort.

## Input Arguments

### **dataElement** — Data element or function argument

data element object | function argument object

Data element or function argument, specified as a `systemcomposer.interface.DataElement` or `systemcomposer.interface.FunctionArgument` object.

### **Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `createOwnedType(dataElement,DataType="double",Dimensions="2",Units="m/s",Complexity="complex",Minimum="0",Maximum="100",Description="Maintain altitude")`

### **DataType** — Data type

character vector | string

Data type, specified as a character vector or string for a valid MATLAB data type. The default value is `double`.

Example: `createOwnedType(dataElement,DataType="double")`

Data Types: `char` | `string`

### **Dimensions** — Dimensions of value type

character vector | string

Dimensions of value type, specified as a character vector or string. The default value is 1.

Example: `createOwnedType(dataElement,Dimensions="2")`

Data Types: `char` | `string`

### **Units** — Units of value type

character vector | string

Units of value type, specified as a character vector or string.

Example: `createOwnedType(dataElement,Units="m/s")`

Data Types: `char` | `string`

### **Complexity** — Complexity of value type

character vector | string

Complexity of value type, specified as a character vector or string. The default value is `real`. Other possible values are `complex` and `auto`.

Example: `createOwnedType(dataElement,Complexity="complex")`

Data Types: char | string

### Minimum — Minimum of value type

character vector | string

Minimum of value type, specified as a character vector or string.

Example: `createOwnedType(dataElement,Minimum="0")`

Data Types: char | string

### Maximum — Maximum of value type

character vector | string

Maximum of value type, specified as a character vector or string.

Example: `createOwnedType(dataElement,Maximum="100")`

Data Types: char | string

### Description — Description of value type

character vector | string

Description of value type, specified as a character vector or string.

Example: `createOwnedType(dataElement,Description="Maintain altitude")`

Data Types: char | string

## Output Arguments

### ownedType — Owned value type

value type object

Owned value type, returned as a `systemcomposer.ValueType` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”



Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`addValueType` | `createModel` | `addInterface` | `setType` | `addServiceInterface` | `createInterface` | `removeInterface`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## systemcomposer.createModel

Create System Composer model

### Syntax

```
model = systemcomposer.createModel(modelName)
model = systemcomposer.createModel(modelName,openFlag)
model = systemcomposer.createModel(modelName,modelType,openFlag)
```

### Description

`model = systemcomposer.createModel(modelName)` creates a System Composer model with name `modelName` and returns the `systemcomposer.arch.Model` object.

`model = systemcomposer.createModel(modelName,openFlag)` creates a System Composer model with name `modelName` and returns the `systemcomposer.arch.Model` object. This function opens the model according to the value of the optional argument `openFlag`.

`model = systemcomposer.createModel(modelName,modelType,openFlag)` creates a System Composer model with name `modelName` and type `modelType` and returns the `systemcomposer.arch.Model` object. This function opens the model according to the value of optional argument `openFlag`.

### Examples

#### Create Model

Create a model, open it, and display its properties.

```
model = systemcomposer.createModel("model_name",true)
```

```
model =
```

```
    model with properties:
```

```
        Name: 'model_name'
    Architecture: [1x1 systemcomposer.arch.Architecture]
    SimulinkHandle: 2.0005
        Views: [0x0 systemcomposer.view.ViewArchitecture]
        Profiles: [0x0 systemcomposer.profile.Profile]
    InterfaceDictionary: [1x1 systemcomposer.interface.Dictionary]
```

### Input Arguments

#### modelName — Name of new model

character vector | string

Name of new model, specified as a character vector or string. This name must be a valid MATLAB identifier.

Example: "model\_name"

Data Types: char | string

### **openFlag – Whether to open model**

false or 0 (default) | true or 1

Whether to open model upon creation, specified as a logical.

Data Types: logical

### **modelType – Type of model**

"Architecture" (default) | "SoftwareArchitecture"

Type of model to create, specified as "Architecture" for an architecture model or "SoftwareArchitecture" for a software architecture model.

Data Types: char | string

## **Output Arguments**

### **model – Architecture model**

model object

Architecture model, returned as a `systemcomposer.arch.Model` object.

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• "Compose Architectures Visually"</li> <li>• "Author Parameters in System Composer Using Parameter Editor"</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>• “Author Software Architectures”</li> <li>• “Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	"Class Diagram View of Software Architectures"

## **Version History**

**Introduced in R2019a**

### **See Also**

open | loadModel | save

### **Topics**

“Compose Architectures Visually”

## systemcomposer.profile.Profile.createProfile

Create profile

### Syntax

```
profile = systemcomposer.profile.Profile.createProfile(profileName)
```

### Description

`profile = systemcomposer.profile.Profile.createProfile(profileName)` creates a new profile with name `profileName`.

---

**Note** Before you move, copy, or rename a profile to a different directory, you must close the profile in the **Profile Editor** or by using the `close` function. If you rename a profile, follow the example for the `renameProfile` function.

---

### Examples

#### Create Profile

Create a model.

```
model = systemcomposer.createModel("archModel");
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");  
latencybase = profile.addStereotype("LatencyBase");  
latencybase.addProperty("latency",Type="double");  
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");  
systemcomposer.profile.editor(profile)  
model.applyProfile("LatencyProfile");
```

Save the profile in a file in the current directory as `LatencyProfile.xml`.

```
path = profile.save;
```

### Input Arguments

#### **profileName** — Name of profile

character vector | string

Name of profile, specified as a character vector or string. Profile must be available on the MATLAB path with a `.xml` extension.

Example: "LatencyProfile"

Data Types: char | string



## Output Arguments

### profile — Profile

profile object

Profile, returned as a `systemcomposer.profile.Profile` object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## **Version History**

**Introduced in R2019a**

### **See Also**

`applyProfile` | `loadProfile` | `editor` | `removeProfile` | `save` | `load` | `open` | `find`

### **Topics**

“Create a Profile and Add Stereotypes”

# createScenario

**Package:** systemcomposer.allocation

Create new empty allocation scenario

## Syntax

```
scenario = createScenario(allocSet,name)
```

## Description

`scenario = createScenario(allocSet,name)` creates a new empty allocation scenario in the allocation set `allocSet` with the given name `name`.

## Examples

### Create Allocation Set and Create New Scenario

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Create a new allocation scenario.

```
newScenario = createScenario(allocSet,"Scenario 2");
```

Open the **Allocation Editor**.

```
systemcomposer.allocation.editor
```

## Input Arguments

### **allocSet** — Allocation set

allocation set object

Allocation set, specified as a `systemcomposer.allocation.AllocationSet` object.

### **name** — Name of allocation scenario

character vector | string

Name of allocation scenario, specified as a character vector or string.

Example: "Scenario 1"

Data Types: char | string

## Output Arguments

### scenario – New empty allocation scenario

allocation scenario object

New empty allocation scenario, returned as a `systemcomposer.allocation.AllocationScenario` object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

`deleteScenario` | `getScenario` | `synchronizeChanges` | `load` | `closeAll` | `find` | `close`

### Topics

“Create and Manage Allocations Programmatically”

# createSimulinkBehavior

**Package:** systemcomposer.arch

Create Simulink behavior and link to component

## Syntax

```
createSimulinkBehavior(component,modelName)
createSimulinkBehavior(component,modelName,"Type",type)
createSimulinkBehavior(component,"Type",type)
createSimulinkBehavior(component,modelName,"BehaviorType",behavior)
```

## Description

`createSimulinkBehavior(component,modelName)` creates a new Simulink model, `modelName`, with the same interfaces as the component `component` and links the component to the new model. The component must have no children.

---

**Note** Components with physical ports cannot be saved as architecture models, model references, software architectures, or Stateflow chart behaviors. Components with physical ports can only be saved as subsystem references or subsystem component behaviors.

---

If no functions are present in software architectures, this syntax creates a rate-based behavior. If functions are present, the syntax creates an export-function behavior.

`createSimulinkBehavior(component,modelName,"Type",type)` creates a new Simulink model or subsystem behavior, `modelName`, with the same interfaces as the component `component` and links the component to the new model. For more information, see "Create Referenced Simulink Behavior Model".

Use this syntax to convert a subsystem component to a subsystem reference.

`createSimulinkBehavior(component,"Type",type)` creates a subsystem component behavior that is part of the parent model. The connections, interfaces, requirement links, and stereotypes of the component are preserved. The component must have no subcomponents and must not already be linked to a model. For more information, see "Create Simulink Subsystem Behavior Using Subsystem Component".

`createSimulinkBehavior(component,modelName,"BehaviorType",behavior)` creates a new Simulink rate-based or export-function behavior, `modelName`, and links the software component to the new model. You can create rate-based or export-function behaviors for software architectures.

## Examples

### **Create Simulink Model and Link to Component**

Create a Simulink model behavior for the component `robotComp` in `Robot.slx` and link the model file to the component.

Create a model `archModel`.

```
model = systemcomposer.createModel("archModel",true);  
arch = get(model,"Architecture");
```

Add two components to the model `electricComp` and `robotComp`. Rearrange the model.

```
names = ["electricComp","robotComp"];  
comp = addComponent(arch,names);  
Simulink.BlockDiagram.arrangeSystem("archModel")
```

Create a Simulink behavior model for the `robotComp` component so the component references the Simulink model `Robot.slx`.

```
createSimulinkBehavior(comp(2),"Robot")
```

### **Create Subsystem Reference Component**

Create a Simulink subsystem behavior for the component `robotComp` in `Robot.slx` and link the subsystem file to the component.

Create a model `archModel`.

```
model = systemcomposer.createModel("archModel",true);  
arch = get(model,"Architecture");
```

Add two components to the model `electricComp` and `robotComp`. Rearrange the model.

```
names = ["electricComp","robotComp"];  
comp = addComponent(arch,names);  
Simulink.BlockDiagram.arrangeSystem("archModel")
```

Create a Simulink subsystem reference behavior for the `robotComp` component so the component references the Simulink subsystem `Robot.slx`.

```
createSimulinkBehavior(comp(2),"Robot",Type="SubsystemReference")
```

### **Create Subsystem Component Behavior and Convert to Subsystem Reference**

Create a Simulink subsystem behavior for the component `robotComp` in `Robot.slx` and link the subsystem file to the component.

Create a model `archModel`.

```
model = systemcomposer.createModel("archModel",true);  
arch = get(model,"Architecture");
```

Add two components to the model `electricComp` and `robotComp`. Rearrange the model.

```
names = ["electricComp", "robotComp"];
comp = addComponent(arch, names);
Simulink.BlockDiagram.arrangeSystem("archModel")
```

Create a Simulink subsystem component behavior for the robotComp component that is part of the parent model.

```
createSimulinkBehavior(comp(2), Type="Subsystem")
```

Convert the subsystem component to a subsystem reference component behavior so the component references the Simulink subsystem Robot.slx.

```
createSimulinkBehavior(comp(2), "Robot", Type="SubsystemReference")
```

### Create Simulink Model with Export-Function Behavior and Link to Software Component

Create a Simulink model with export-function behavior myBehaviorModel.slx for the software component named C1 and link the model to the component.

Create a software architecture model named mySoftwareModel.

```
model=systemcomposer.createModel("mySoftwareModel", "SoftwareArchitecture", true);
arch = get(model, "Architecture");
```

Add a component C1 to the model.

```
comp = addComponent(arch, "C1");
```

Create a Simulink model with an export-function behavior named myBehaviorModel.slx that is referenced by the component C1.

```
createSimulinkBehavior(comp, "myBehaviorModel", BehaviorType="ExportFunction")
```

## Input Arguments

### **component** — System or software architecture component

component object

System or software architecture component with no children, specified as a `systemcomposer.arch.Component` object. This component can also be specified as a subsystem component to be converted to a subsystem reference.

### **modelName** — Name of model

character vector | string

Name of model, specified as a character vector or string.

Example: "exMobileRobot"

Data Types: char | string

### **behavior** — Component behavior

"RateBased" | "ExportFunction"

Component behavior, specified as one of these values:

- "RateBased" to create a rate-based component behavior
- "ExportFunction" to create an export-function component behavior

Data Types: char | string

### type – Component behavior

"ModelReference" | "SubsystemReference" | "Subsystem"

Component behavior, specified as one of these values:

- "ModelReference" to create a Simulink model reference component behavior
- "SubsystemReference" to create a Simulink subsystem reference component behavior
- "Subsystem" to create a Simulink subsystem component behavior

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”



Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>• “Author Software Architectures”</li> <li>• “Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

## Version History

Introduced in R2019a

### See Also

[inlineComponent](#) | [createArchitectureModel](#) | [createStateflowChartBehavior](#) | [extractArchitectureFromSimulink](#) | [linkToModel](#) | [isReference](#) | [Reference Component](#)

### Topics

["Implement Component Behavior Using Simulink"](#)  
["Decompose and Reuse Components"](#)  
["Implement Component Behavior Using Stateflow Charts"](#)  
["Create Simulink Subsystem Behavior Using Subsystem Component"](#)  
["Simulate and Deploy Software Architectures"](#)

# createStateflowChartBehavior

**Package:** systemcomposer.arch

Add Stateflow chart behavior to component

## Syntax

```
createStateflowChartBehavior(component)
```

## Description

`createStateflowChartBehavior(component)` adds Stateflow Chart behavior to a component component. The connections, interfaces, requirement links, and stereotypes are preserved. The component must have no subcomponents and must not already be linked to a model.

---

**Note** Components with physical ports cannot be saved as architecture models, model references, software architectures, or Stateflow chart behaviors. Components with physical ports can only be saved as subsystem references or subsystem component behaviors.

---

## Examples

### Add Stateflow Chart Behavior to Component

Add Stateflow chart behavior to the component named "robotComp" within the current model.

Create a model named "archModel".

```
model = systemcomposer.createModel("archModel", true);  
arch = get(model, "Architecture");
```

Add two components to the model with the names "electricComp" and "robotComp". Rearrange the model.

```
names = ["electricComp", "robotComp"];  
comp = addComponent(arch, names);  
Simulink.BlockDiagram.arrangeSystem("archModel")
```

Add Stateflow chart behavior to the robotComp component.

```
createStateflowChartBehavior(comp(2));
```

## Input Arguments

### component — Component

component object

Component with no subcomponents, specified as a `systemcomposer.arch.Component` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>



Term	Definition	Application	More Information
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>“Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>“Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2021a

### See Also

[inlineComponent](#) | [createSimulinkBehavior](#) | [createArchitectureModel](#) | [extractArchitectureFromSimulink](#) | [linkToModel](#) | [isReference](#) | [Reference Component](#)

### Topics

“Implement Component Behavior Using Simulink”  
 “Decompose and Reuse Components”  
 “Implement Component Behavior Using Stateflow Charts”  
 “Create Simulink Subsystem Behavior Using Subsystem Component”  
 “Simulate and Deploy Software Architectures”

# createSubsystemBehavior

**Package:** `systemcomposer.arch`

Add subsystem behavior to component

---

**Note** The `createSubsystemBehavior` function is not recommended. Use the `createSimulinkBehavior` function instead. For more information, see “Compatibility Considerations”.

---

## Syntax

```
createSubsystemBehavior(component)
```

## Description

`createSubsystemBehavior(component)` adds subsystem behavior to the component `component`. The connections, interfaces, requirement links, and stereotypes of the component are preserved. The component must have no subcomponents and must not already be linked to a model.

## Input Arguments

**component** — **Component**

`component` object

Component with no subcomponents, specified as a `systemcomposer.arch.Component` object.

## Version History

**Introduced in R2021b**

**R2022a\_plus: createSubsystemBehavior function is not recommended**

*Not recommended starting in R2022a\_plus*

The `createSubsystemBehavior` function is not recommended. Use the `createSimulinkBehavior` function instead.

## See Also

`inlineComponent` | `createSimulinkBehavior` | `createArchitectureModel` | `createStateflowChartBehavior` | `extractArchitectureFromSimulink` | `linkToModel` | `isReference` | Reference Component

## Topics

“Implement Component Behavior Using Simulink”

“Decompose and Reuse Components”

“Implement Component Behavior Using Stateflow Charts”

“Create Simulink Subsystem Behavior Using Subsystem Component”

“Simulate and Deploy Software Architectures”

## createSubGroup

**Package:** systemcomposer.view

Create subgroup in element group of view

### Syntax

```
subGroup = createSubGroup(elementGroup, subGroupName)
```

### Description

`subGroup = createSubGroup(elementGroup, subGroupName)` creates a new subgroup `subGroup`, named `subGroupName` within the element group `elementGroup` of an architecture view.

---

**Note** This function cannot be used when a selection query or grouping is defined on the view. To remove the query, run `removeQuery`.

---

### Examples

#### Create Subgroup in View

Open the keyless entry system example and create a view `newView`.

```
sckeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("newView");
```

Open the Architecture Views Gallery to see the new view `newView`.

```
model.openViews
```

Create a subgroup `myGroup`.

```
group = view.Root.createSubGroup("myGroup")
```

```
group =
  ElementGroup with properties:
      Name: 'myGroup'
      UUID: 'cd8a3f23-db62-498f-8d41-d04cb4561e78'
      Elements: []
      SubGroups: [0x0 systemcomposer.view.ElementGroup]
```

### Input Arguments

**elementGroup — Element group**

element group object

Element group for view, specified as a `systemcomposer.view.ElementGroup` object.

### **subGroupName — Name of subgroup**

character vector | string

Name of subgroup, specified as a character vector or string.

Example: "myGroup"

Data Types: char | string

## **Output Arguments**

### **subGroup — Subgroup**

element group object

Subgroup, returned as a `systemcomposer.view.ElementGroup` object.

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"

Term	Definition	Application	More Information
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

openViews | createView | getView | deleteView | systemcomposer.view.ElementGroup | systemcomposer.view.View | getSubGroup | deleteSubGroup | addElement | removeElement

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# createView

**Package:** systemcomposer.arch

Create architecture view

## Syntax

```
view = createView(model,name)
view = createView( ____,Name,Value)
```

## Description

`view = createView(model,name)` creates a new architecture view `view` for the System Composer model `model` with the specified name `name`.

To delete a view, use the `deleteView` function.

`view = createView( ____,Name,Value)` creates a new view with additional options.

## Examples

### Create View with Query and Group By

Open the keyless entry system example and create a view. Specify the color as light blue and the query as all components, and group by the review status.

```
sckKeylessEntrySystem
import systemcomposer.query.*
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("All Components Grouped by Review Status",...
    Color="lightblue",Select=AnyComponent,...
    GroupBy="AutoProfile.BaseComponent.ReviewStatus");
```

Open the Architecture Views Gallery to see the new view named All Components Grouped by Review Status.

```
model.openViews
```

## Input Arguments

### **model** — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

### **name** — Name of view

character vector | string

Name of view, specified as a character vector or string.

Example: "All Components Grouped by Review Status"

Data Types: char | string

### Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `view = model.createView("All Components Grouped by Review Status",Color="lightblue",Select=AnyComponent(),GroupBy="AutoProfile.BaseComponent.ReviewStatus")`

### Select — Selection query

constraint object

Selection query to use to populate the view, specified as a `systemcomposer.query.Constraint` object.

A constraint can contain a subconstraint that can be joined with another constraint using `AND` or `OR`. A constraint can be negated using `NOT`.

Example: `view = model.createView("All Components Grouped by Review Status",Select=HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent")))`

### Query Objects and Conditions for Constraints

Query Object	Condition
Property	A non-evaluated value for the given property or stereotype property.
PropertyValue	An evaluated property value from a System Composer object or a stereotype property.
HasConnector	A component has a connector that satisfies the given subconstraint.
HasPort	A component has a port that satisfies the given subconstraint.
HasInterface	A port has an interface that satisfies the given subconstraint.
HasInterfaceElement	An interface has an interface element that satisfies the given subconstraint.
HasStereotype	An architecture element has a stereotype that satisfies the given subconstraint.
IsInRange	A property value is within the given range.
AnyComponent	An element is a component and not a port or connector.
IsStereotypeDerivedFrom	A stereotype is derived from the given stereotype.



**GroupBy — Grouping criteria**

cell array of character vectors | array of strings

Grouping criteria, specified as a cell array of character vectors or an array of strings in the form "`<profile>.<stereotype>.<property>`". The order of the cell array dictates the order of the grouping.

```
Example: view = model.createView("All Components Grouped by Review  
Status", GroupBy=["AutoProfile.MechanicalComponent.mass", "AutoProfile.Mechanica  
lComponent.cost"])
```

Data Types: char | string

**IncludeReferenceModels — Whether to search for reference architectures**

true or 1 (default) | false or 0

Whether to search for reference architectures, specified as a logical.

```
Example: view = model.createView("All Components Grouped by Review  
Status", IncludeReferenceModels=false)
```

Data Types: logical

**Color — Color of view**

character vector | string

Color of view, specified as a character vector or string that contains the name of the color or an RGB hexadecimal value.

```
Example: view = model.createView("All Components Grouped by Review  
Status", Color="blue")
```

```
Example: view = model.createView("All Components Grouped by Review  
Status", Color="#FF00FF")
```

Data Types: char | string

**Output Arguments****view — Architecture view**

view object

Architecture view, returned as a `systemcomposer.view.View` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"

Term	Definition	Application	More Information
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

`systemcomposer.view.View` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup` | `getQualifiedName`

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# createViewArchitecture

**Package:** systemcomposer.arch

(Removed) Create view

---

**Note** The `createViewArchitecture` function has been removed. You can create a view using the `createView` function. For further details, see “Compatibility Considerations”.

---

## Syntax

```
view = createViewArchitecture(model, name)
view = createViewArchitecture(model, name, constraint)
view = createViewArchitecture(model, name, constraint, groupBy)
view = createViewArchitecture( ____, Name, Value)
```

## Description

`view = createViewArchitecture(model, name)` creates an empty view with the given name and default color 'blue'.

`view = createViewArchitecture(model, name, constraint)` creates a view with the given name where the contents are populated by finding all components in the model that satisfy the provided query.

`view = createViewArchitecture(model, name, constraint, groupBy)` creates a view with the given name where the contents are populated by finding all components in the model that satisfy the provided query. The selected components are then grouped by the fully qualified property name.

`view = createViewArchitecture( ____, Name, Value)` creates a view with additional options.

## Examples

### Create View Based on Query and Group By Review Status

```
scKeylessEntrySystem;
m = systemcomposer.openModel('KeylessEntryArchitecture');

import systemcomposer.query.*;
myQuery = HasStereotype(IsStereotypeDerivedFrom('AutoProfile.SoftwareComponent'));

view = m.createViewArchitecture('Software Review Status', myQuery, ...
    'AutoProfile.BaseComponent.ReviewStatus', 'Color', 'red');

m.openViews;
```

## Input Arguments

**model** — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

**name — Name of view**

character vector

Name of view, specified as a character vector.

Data Types: char

**constraint — Query**

query constraint object

Query, specified as a `systemcomposer.query.Constraint` object representing specific conditions.

A constraint can contain a subconstraint that can be joined with another constraint using AND or OR. A constraint can be negated using NOT.

**Query Objects and Conditions for Constraints**

Query Object	Condition
Property	A non-evaluated value for the given property or stereotype property.
PropertyValue	An evaluated property value from a System Composer object or a stereotype property.
HasConnector	A component has a connector that satisfies the given subconstraint.
HasPort	A component has a port that satisfies the given subconstraint.
HasInterface	A port has an interface that satisfies the given subconstraint.
HasInterfaceElement	An interface has an interface element that satisfies the given subconstraint.
HasStereotype	An architecture element has a stereotype that satisfies the given subconstraint.
IsInRange	A property value is within the given range.
AnyComponent	An element is a component and not a port or connector.
IsStereotypeDerivedFrom	A stereotype is derived from the given stereotype.

**groupBy — User-defined property**

enumeration

User-defined property, specified as an enumeration by which to group components.

Data Types: enum

**Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

```
Example: createViewArchitecture(model, 'Software Review
Status', myQuery, 'AutoProfile.BaseComponent.ReviewStatus', 'Color', 'red', 'IncludeReferenceModels', true)
```

### **IncludeReferenceModels — Whether to search for reference architectures**

false or 0 (default) | true or 1

Whether to search for reference architectures, or to not include referenced architectures, specified as the comma-separated pair consisting of 'IncludeReferenceModels' and a logical 0 (false) to not include referenced architectures and 1 (true) to search for referenced architectures.

Example: 'IncludeReferenceModels', true

Data Types: logical

### **Color — Color of view**

character array

Color of view, specified as the comma-separated pair consisting of 'Color' and a character array that contains the name of the color or an RGB hexadecimal value.

Example: 'Color', 'blue'

Example: 'Color', '#FF00FF'

Data Types: char

## **Output Arguments**

### **view — Model architecture view**

view architecture object

Model architecture view created based on the specified query and properties, returned as a `systemcomposer.view.ViewArchitecture` object.

## **Version History**

### **Introduced in R2019b**

### **R2021a: createViewArchitecture function has been removed**

*Errors starting in R2021a*

The `createViewArchitecture` function is removed in R2021a with the introduction of a new set of views API. For more information on how to create and edit a view using the command line, see “Create Architectural Views Programmatically”.

## **See Also**

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup`

## **Topics**

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# createViewComponent

**Package:** systemcomposer.view

(Removed) Create view component

---

**Note** The createViewComponent function has been removed. You can create a view using the createView function and then add a component using the addElement function. Add a subgroup with the createSubGroup function. For further details, see “Compatibility Considerations”.

---

## Syntax

```
viewComp = createViewComponent(object,name)
```

## Description

viewComp = createViewComponent(object,name) creates a new view component with the provided name.

createViewComponent is a method for the class systemcomposer.view.ViewArchitecture.

## Examples

### Create View Component

Create view component with context view.

```
scKeylessEntrySystem
zcModel = systemcomposer.loadModel('KeylessEntryArchitecture');
fobSupplierView = zcModel.createViewArchitecture("FOB Locator System Supplier Breakdown",...
    "Color","lightblue");
supplierD = fobSupplierView.createViewComponent("Supplier D");
```

## Input Arguments

### object — View architecture

view architecture object

View architecture, specified as a systemcomposer.view.ViewArchitecture object.

### name — Name of component

character vector

Name of component, specified as a character vector.

Data Types: char



## Output Arguments

### **viewComp — View component**

view component object

View component, returned as a `systemcomposer.view.ViewComponent` object.

## Version History

**Introduced in R2019b**

### **R2021a: createViewComponent function has been removed**

*Errors starting in R2021a*

The `createViewComponent` function is removed in R2021a with the introduction of a new set of views API. For more information on how to create and edit a view using the command line, see “Create Architectural Views Programmatically”.

## See Also

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup`

## Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# deallocate

**Package:** systemcomposer.allocation

Delete allocation

## Syntax

```
deallocate(allocScenario, sourceElement, targetElement)
```

## Description

`deallocate(allocScenario, sourceElement, targetElement)` deletes allocation, if one exists, between the source element `sourceElement` and the target element `targetElement`.

## Examples

### Create Allocation Set and Deallocate Elements Between Models

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario,sourceComp,targetComp);
```

Deallocate components between models.

```
deallocate(defaultScenario,sourceComp,targetComp);
```

Save the allocation set.

```
save(allocSet)
```

Open the **Allocation Editor**.

```
systemcomposer.allocation.editor
```

## Input Arguments

**allocScenario** — Allocation scenario

allocation scenario object

Allocation scenario , specified as a `systemcomposer.allocation.AllocationScenario` object.

### sourceElement — Source element

element object

Source element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

### targetElement — Target element

element object

Target element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

**See Also**

getAllocation | getAllocatedFrom | allocate | getAllocatedTo | destroy | getScenario | createAllocationSet

**Topics**

“Create and Manage Allocations Programmatically”

# decreaseExecutionOrder

**Package:** systemcomposer.arch

Change function execution order to earlier

## Syntax

```
decreaseExecutionOrder(functionObj)
```

## Description

`decreaseExecutionOrder(functionObj)` decreases execution order of the specified function `functionObj` by 1. If the function is at the minimum execution order, the `decreaseExecutionOrder` method will fail with a warning.

## Examples

### Change Execution Order of Software Functions

This example shows the software architecture of a throttle position control system and how to schedule the execution order of the root level functions.

```
model = systemcomposer.openModel("ThrottleControlComposition");
```

Simulate the model to populate it with functions.

```
sim("ThrottleControlComposition");
```

View the function names ordered by execution order.

```
functions = {model.Architecture.Functions.Name}'
```

```
functions = 6x1 cell
    {'Actuator_output_5ms' }
    {'Controller_run_5ms' }
    {'TPS_Primary_read_5ms' }
    {'TPS_Secondary_read_5ms' }
    {'TP_Monitor_D1' }
    {'APP_Sensor_read_10ms' }
```

Decrease the execution order of the third function.

```
decreaseExecutionOrder(model.Architecture.Functions(3))
```

View the function names ordered by execution order.

```
functions = {model.Architecture.Functions.Name}'
```

```
functions = 6x1 cell
    {'Actuator_output_5ms' }
    {'TPS_Primary_read_5ms' }
```

```

{'Controller_run_5ms'   }
{'TPS_Secondary_read_5ms'}
{'TP_Monitor_D1'      }
{'APP_Sensor_read_10ms'}

```

The third function is now moved up in execution order, executing earlier.

Increase the execution order of the second function.

```
increaseExecutionOrder(model.Architecture.Functions(2))
```

View the function names ordered by execution order.

```

functions = {model.Architecture.Functions.Name}'

functions = 6x1 cell
    {'Actuator_output_5ms'   }
    {'Controller_run_5ms'   }
    {'TPS_Primary_read_5ms' }
    {'TPS_Secondary_read_5ms'}
    {'TP_Monitor_D1'      }
    {'APP_Sensor_read_10ms'}

```

The second function is now moved down in execution order, executing later.

## Input Arguments

### functionObj – Function

function object

Function, specified as a `systemcomposer.arch.Function` object.

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>

Term	Definition	Application	More Information
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”



## **Version History**

**Introduced in R2021b**

### **See Also**

`systemcomposer.createModel` | `createArchitectureModel` | `increaseExecutionOrder`

### **Topics**

“Modeling Software Architecture of Throttle Position Control System”

“Simulate and Deploy Software Architectures”

“Author Software Architectures”

## systemcomposer.analysis.deleteInstance

Delete architecture instance

### Syntax

```
systemcomposer.analysis.deleteInstance(instance)
```

### Description

systemcomposer.analysis.deleteInstance(instance) deletes an existing instance.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

### Examples

#### Delete Architecture Instance

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Instantiate all stereotypes in the profile.

```
model = systemcomposer.createModel("archModel",true);
instance = instantiate(model.Architecture,"LatencyProfile","NewInstance");
```

Delete the architecture instance.

```
systemcomposer.analysis.deleteInstance(instance);
```

## Input Arguments

### **instance** — Architecture instance

architecture instance object

Architecture instance to be deleted, specified as a `systemcomposer.analysis.ArchitectureInstance` object.

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

## **Version History**

**Introduced in R2019a**

### **See Also**

`instantiate` | `systemcomposer.analysis.Instance` | `loadInstance` | `save` | `refresh` | `update`

### **Topics**

“Write Analysis Function”

# deleteScenario

**Package:** systemcomposer.allocation

Delete allocation scenario

## Syntax

```
deleteScenario(allocSet, name)
```

## Description

deleteScenario(allocSet, name) deletes the allocation scenario in the set allocSet with the given name name.

## Examples

### Create Allocation Set and Delete Scenario

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation", true);
sourceComp = addComponent(get(mSource, "Architecture"), "Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation", true);
targetComp = addComponent(get(mTarget, "Architecture"), "Target_Component");
```

Create the allocation set MyNewAllocation.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation", ...
    "Source_Model_Allocation", "Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet, "Scenario 1");
```

Create a new allocation scenario.

```
newScenario = createScenario(allocSet, "Scenario 2");
```

Delete the default allocation scenario.

```
deleteScenario(allocSet, "Scenario 1");
```

Save the allocation set.

```
save(allocSet)
```

Open the **Allocation Editor**.

```
systemcomposer.allocation.editor
```

## Input Arguments

**allocSet** — Allocation set

allocation set object

Allocation set, specified as a `systemcomposer.allocation.AllocationSet` object.

**name — Name of allocation scenario**

character vector | string

Name of allocation scenario, specified as a character vector or string.

Example: "Scenario 1"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

`getScenario` | `createScenario` | `synchronizeChanges` | `load` | `closeAll` | `find` | `close`

### Topics

“Create and Manage Allocations Programmatically”

# deleteSubGroup

**Package:** `systemcomposer.view`

Delete subgroup in element group of view

## Syntax

```
deleteSubGroup(elementGroup, subGroupName)
```

## Description

`deleteSubGroup(elementGroup, subGroupName)` deletes the subgroup named `subGroupName` within the element group `elementGroup` of an architecture view.

## Examples

### Create and Delete Subgroup in View

Open the keyless entry system example and create a view `newView`.

```
scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("newView");
```

Open the Architecture Views Gallery to see the new view `newView`.

```
model.openViews
```

Create a subgroup `myGroup`.

```
group = view.Root.createSubGroup("myGroup");
```

Delete the subgroup `myGroup`.

```
view.Root.deleteSubGroup("myGroup");
```

## Input Arguments

### **elementGroup** — Element group

element group object

Element group for view, specified as a `systemcomposer.view.ElementGroup` object.

### **subGroupName** — Name of subgroup

character vector | string

Name of subgroup, specified as a character vector or string.

Example: `"myGroup"`

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”



Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

openViews | createView | getView | deleteView | systemcomposer.view.ElementGroup | systemcomposer.view.View | getSubGroup | createSubGroup | removeElement | addElement

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

## deleteView

**Package:** `systemcomposer.arch`

Delete architecture view

### Syntax

```
deleteView(model, name)
```

### Description

`deleteView(model, name)` deletes the view `name`, if it exists, in the specified model `model`.

### Examples

#### Create and Delete View

Open the keyless entry system example and create a view, `newView`.

```
scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("newView");
```

Open the Architecture Views Gallery to see `newView`.

```
model.openViews
```

Delete the view and see that it has been deleted.

```
model.deleteView("newView")
```

### Input Arguments

#### **model** — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

#### **name** — Name of view

character vector | string

Name of view, specified as a character vector or string.

Example: "All Components Grouped by Review Status"

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"

Term	Definition	Application	More Information
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

`systemcomposer.view.View` | `openViews` | `getView` | `createView` | `systemcomposer.view.ElementGroup`

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# destroy

**Package:** `systemcomposer.arch`

Remove model element

## Syntax

```
destroy(element)
```

## Description

`destroy(element)` removes and destroys the architecture model element `element`.

## Examples

### Destroy Component

Create a component, `newComponent`, then remove it from the model.

```
model = systemcomposer.createModel("newModel", true);
rootArch = get(model, "Architecture");
newComponent = addComponent(rootArch, "newComponent");
destroy(newComponent)
```

## Input Arguments

### **element** – Architecture model element

component object | variant component object | architecture port object | connector object | physical connector object | function object | value type object | data interface object | data element object | physical domain object | physical interface object | physical element object | function argument object | service interface object | function element object | property object | view object | element group object | allocation scenario object | allocation object | parameter object

Architecture model element, specified as one of these objects:

- `systemcomposer.arch.Component`
- `systemcomposer.arch.VariantComponent`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.arch.PhysicalConnector`
- `systemcomposer.arch.Function`
- `systemcomposer.ValueType`
- `systemcomposer.interface.DataInterface`
- `systemcomposer.interface.DataElement`
- `systemcomposer.interface.PhysicalDomain`

- `systemcomposer.interface.PhysicalInterface`
- `systemcomposer.interface.PhysicalElement`
- `systemcomposer.interface.FunctionArgument`
- `systemcomposer.interface.ServiceInterface`
- `systemcomposer.interface.FunctionElement`
- `systemcomposer.profile.Property`
- `systemcomposer.view.View`
- `systemcomposer.view.ElementGroup`
- `systemcomposer.allocation.AllocationScenario`
- `systemcomposer.allocation.Allocation`
- `systemcomposer.arch.Parameter`

## Version History

Introduced in R2019a

### See Also

`Component` | `Variant Component` | `removeElement` | `removeElement` | `removeInterface` | `deleteView` | `deleteSubGroup` | `deleteInstance` | `removeProfile` | `removeProperty` | `removeStereotype` | `removeStereotype` | `deallocate` | `deleteScenario`

## systemcomposer.allocation.editor

Open allocation editor

### Syntax

```
systemcomposer.allocation.editor  
systemcomposer.allocation.editor(allocSet)  
systemcomposer.allocation.editor(allocSetName)
```

### Description

`systemcomposer.allocation.editor` opens the **Allocation Editor**.

`systemcomposer.allocation.editor(allocSet)` opens the **Allocation Editor** and selects the allocation set object `allocSet`.

`systemcomposer.allocation.editor(allocSetName)` opens the **Allocation Editor** and selects the allocation set `allocSetName`.

### Examples

#### Create Allocation Set and Open in Allocation Editor

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);  
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");  
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);  
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...  
    "Source_Model_Allocation","Target_Model_Allocation");
```

Save the allocation set.

```
save(allocSet)
```

Open the **Allocation Editor**.

```
systemcomposer.allocation.editor
```

### Input Arguments

#### **allocSet** — Allocation set

allocation set object

Allocation set, specified as a `systemcomposer.allocation.AllocationSet` object.

#### **allocSetName** — Allocation set name

character vector | string



Allocation set name, specified as a character vector or string.

Example: `systemcomposer.allocation.editor("PhysicalAllocations")`

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

`createAllocationSet` | `systemcomposer.allocation.AllocationSet`

### Topics

“Create and Manage Allocations Programmatically”

## systemcomposer.profile.editor

Open Profile Editor

### Syntax

```
systemcomposer.profile.editor  
systemcomposer.profile.editor(profile)  
systemcomposer.profile.editor(profileName)
```

### Description

`systemcomposer.profile.editor` opens the System Composer **Profile Editor**.

`systemcomposer.profile.editor(profile)` opens the **Profile Editor** and selects the profile object `profile`.

`systemcomposer.profile.editor(profileName)` opens the **Profile Editor** and selects the profile `profileName`.

### Examples

#### Open Profile Editor

Create and save a profile, then open the **Profile Editor** with that profile selected.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");  
profile.save  
systemcomposer.profile.editor(profile)
```

### Input Arguments

#### **profile** — Profile

profile object

Profile, specified as a `systemcomposer.profile.Profile` object.

#### **profileName** — Profile name

character vector | string

Profile name, specified as a character vector or string.

Example: `systemcomposer.profile.editor("LatencyProfile")`

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

**See Also**

systemcomposer.profile.Profile | loadProfile | open | load | find | save | closeAll | createProfile

**Topics**

“Define Profiles and Stereotypes”

# systemcomposer.exportModel

Export model information as MATLAB tables

## Syntax

```
[exportedSet] = systemcomposer.exportModel(modelName)
[exportedSet,errorLog] = systemcomposer.exportModel(modelName)
```

## Description

[exportedSet] = systemcomposer.exportModel(modelName) exports model information for components, ports, connectors, port interfaces, and requirement links, with a domain field to be imported into MATLAB tables. For software architectures, the programmatic interface exports function information. The exported tables have prescribed formats to specify model element relationships, stereotypes, and properties. For more information on the import structure, see the `importModel` function and "Import and Export Architecture Models".

[exportedSet,errorLog] = systemcomposer.exportModel(modelName) exports model information to be imported into MATLAB tables with output arguments `exportedSet` with a structure of exported tables and `errorLog` to display export error information.

## Examples

### Export System Composer Model

To export a model, pass the model name as an argument to the `exportModel` function. The function returns a structure containing five tables: `components`, `ports`, `connections`, `portInterfaces`, `requirementLinks`, and `parameters` with a `domain` field returned as 'System' for architecture models and 'Software' for software architecture models.

```
exportedSet = systemcomposer.exportModel('exMobileRobot')
```

```
exportedSet =
```

```
struct with fields:
```

```
    components: [3×4 table]
         ports: [3×5 table]
    connections: [1×4 table]
    portInterfaces: [3×9 table]
    requirementLinks: [4×15 table]
         parameters: [6×9 table]
         domain: 'System'
```

### Export Software Architecture Model

To export a software architecture model, pass the model name as an argument to the `exportModel` function. The function returns a structure containing seven tables: `components`, `ports`,

connections, portInterfaces, requirementLinks, parameters, domain as 'Software', and functions.

```
exportedSet = systemcomposer.exportModel('mySoftwareArchitectureModel')
```

```
exportedSet =
```

```
    struct with fields:
        components: [2×5 table]
        ports: [0×4 table]
        connections: [0×4 table]
        portInterfaces: [0×9 table]
        requirementLinks: [0×15 table]
        parameters: [0×9 table]
        domain: 'Software'
        functions: [1×4 table]
```

## Input Arguments

### **modelName** — Name of model

character vector | string

Name of model, specified as a character vector or string.

Example: "exMobileRobot"

Data Types: char | string

## Output Arguments

### **exportedSet** — Model tables

structure

Model tables, returned as a structure containing tables for components, ports, connections, portInterfaces, requirementLinks, and parameters, with a domain field returned as 'System' for architecture models, and 'Software' for software architecture models. For software architectures, model tables include a functions table for exported function information.

Data Types: struct

### **errorLog** — Errors reported during export process

string array

Errors reported during export process, returned as a string array. You can obtain the error text by calling the disp method on the array of strings. For example, disp(errorLog) is used to obtain the errors reported as strings in a readable format.

Data Types: string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## Version History

Introduced in R2019a

### See Also

#### Blocks

Component | Variant Component | Reference Component

#### Functions

importModel | systemcomposer.exportToVersion |  
systemcomposer.updateLinksToReferenceRequirements

#### Topics

“Import and Export Architecture Models”

“Author Parameters in System Composer Using Parameter Editor”



# systemcomposer.exportToVersion

Export architecture model and dependencies to previous release of System Composer

## Syntax

```
systemcomposer.exportToVersion(modelName, dirName, version)
```

## Description

`systemcomposer.exportToVersion(modelName, dirName, version)` exports an architecture model with name `modelName` and its dependencies to the version of System Composer given by `version`. The exported artifacts are created in a directory specified by `dirName`.

## Examples

### Create Model Then Export to Previous Version

Create an architecture model, then export that model to a previous version of System Composer.

```
model = systemcomposer.createModel("OlderVersionModel");
save(model)
systemcomposer.exportToVersion('OlderVersionModel', "OlderVersion", "R2021a")
```

## Input Arguments

### **modelName** — Name of model

character vector | string

Name of model, specified as a character vector or string.

Example: "exMobileRobot"

Data Types: char | string

### **dirName** — Name of empty directory

character vector | string

Name of empty directory, specified as a character vector or string. You can specify either the relative path to the directory or the full path.

Example: "Projects/Aero"

Data Types: char | string

### **version** — Version of MATLAB

character vector | string

Version of MATLAB, specified as a character vector or string.

Example: "R2022b"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019b

### See Also

#### Blocks

Component | Variant Component | Reference Component

#### Functions

exportModel | importModel | systemcomposer.updateLinksToReferenceRequirements

#### Topics

"Organize System Composer Files in Projects"

"Import and Export Architecture Models"

# systemcomposer.extractArchitectureFromSimulink

Extract architecture from Simulink model

## Syntax

```
systemcomposer.extractArchitectureFromSimulink(model,name)
systemcomposer.extractArchitectureFromSimulink(model,name,Name,Value)
```

## Description

`systemcomposer.extractArchitectureFromSimulink(model,name)` exports the Simulink model `model` to an architecture model `name` and saves it in the current directory.

`systemcomposer.extractArchitectureFromSimulink(model,name,Name,Value)` exports the Simulink model `model` to an architecture model `name` and saves it in the current directory with additional options.

## Examples

### Extract Architecture of Simulink Model Using System Composer

Export an existing Simulink® model to a System Composer™ architecture model. The algorithmic sections of the original model are removed and structural information is preserved during this process. Requirements links, if present, are also preserved.

### Convert Simulink Model to System Composer Architecture

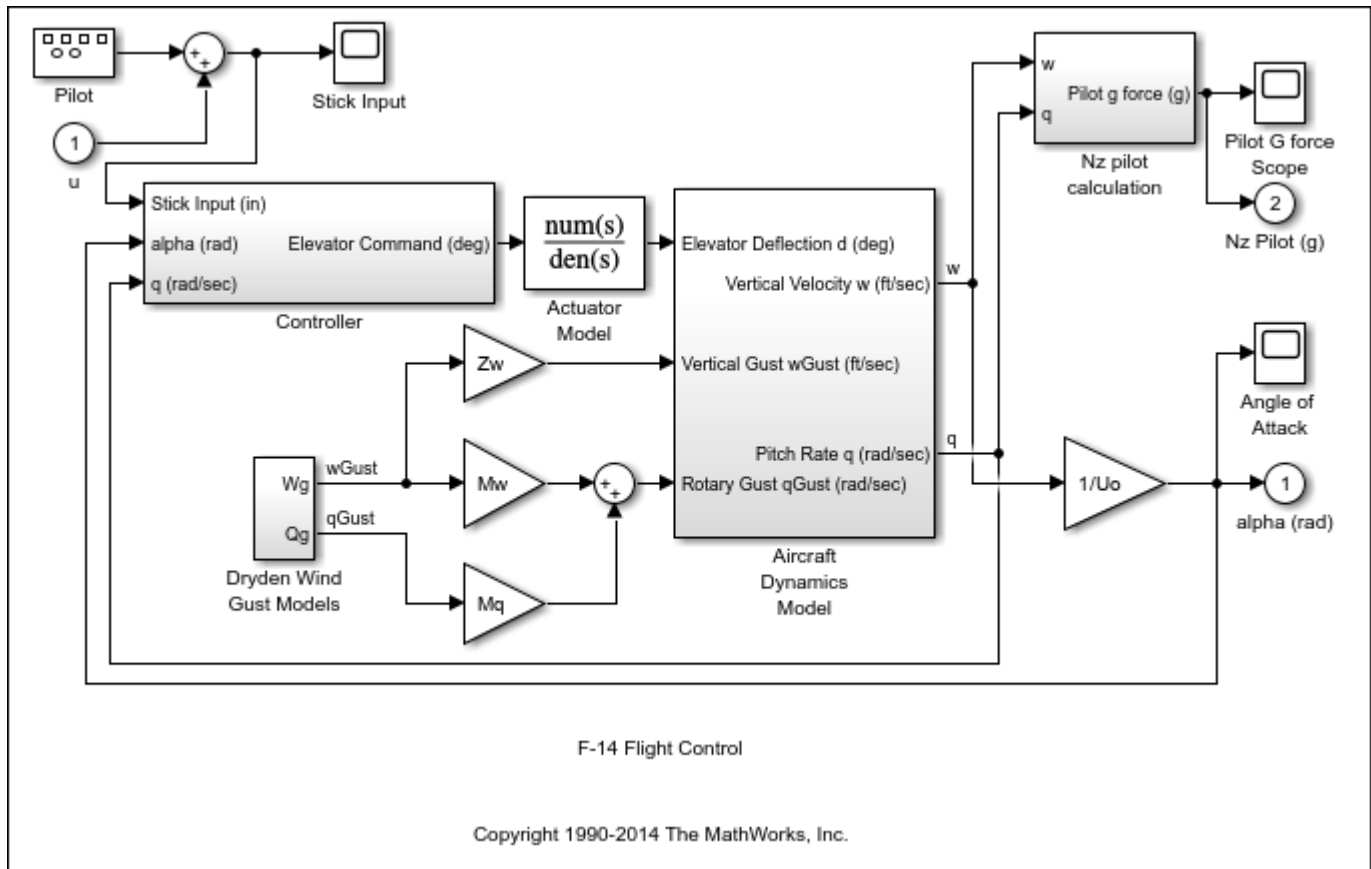
System Composer converts structural constructs in a Simulink model to equivalent architecture model constructs:

- Subsystems to components
- Variant subsystems to variant components
- Bus objects to interfaces
- Referenced models to reference components

### Open Model

Open the Simulink model of F-14 Flight Control.

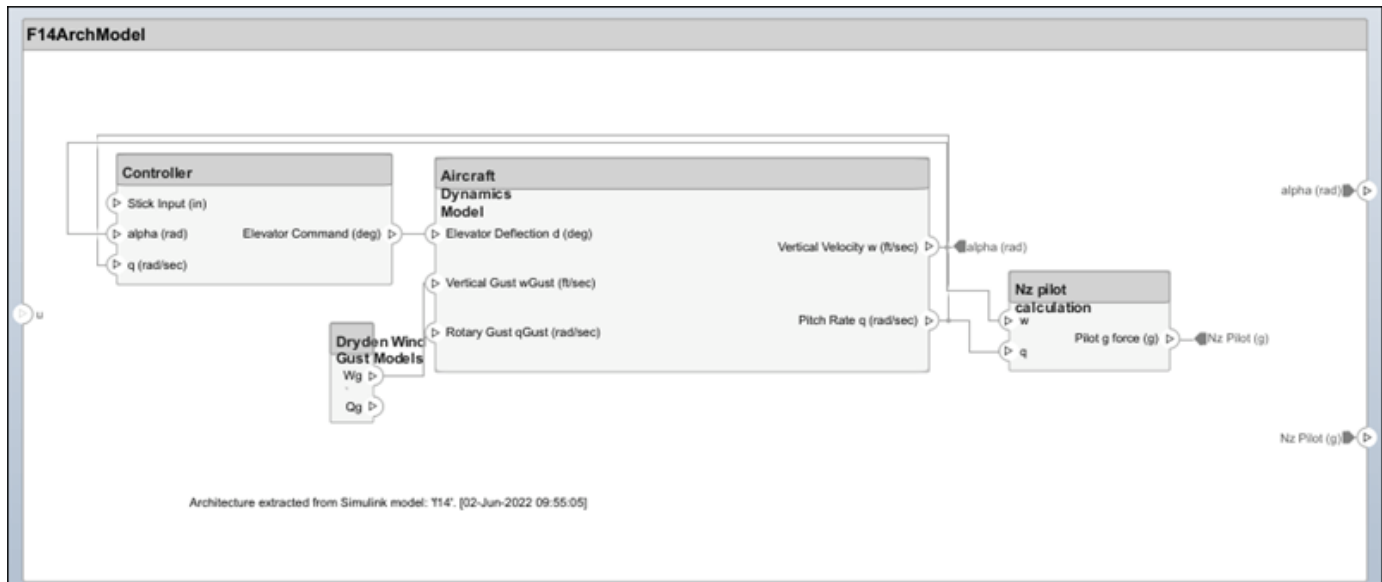
```
open_system('f14')
```



### Export Model

Extract an architecture model from the original model.

```
systemcomposer.extractArchitectureFromSimulink('f14', 'F14ArchModel');
Simulink.BlockDiagram.arrangeSystem('F14ArchModel');
systemcomposer.openModel('F14ArchModel');
```



## Input Arguments

### model — Simulink model name

character vector | string

Simulink model name from which to extract the architecture, specified as a character vector or string. The model must be on the path.

Example: "f14"

Data Types: char | string

### name — Architecture model name

character vector | string

Architecture model name, specified as a character vector or string. This model is saved in the current directory.

Example: "F14ArchModel"

Data Types: char | string

### Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example:

```
systemcomposer.extractArchitectureFromSimulink("f14","F14ArchModel",AutoArrange=false,ShowProgress=true)
```

### AutoArrange — Whether to auto-arrange architecture model

true or 1 (default) | false or 0

Whether to auto-arrange architecture model, specified as a logical.

Example:

```
systemcomposer.extractArchitectureFromSimulink("f14","F14ArchModel",AutoArrange=false)
```

Data Types: logical

### ShowProgress – Whether to show progress bar

false or 0 (default) | true or 1

Whether to show progress bar, specified as a logical. This option is useful for larger models.

Example:

```
systemcomposer.extractArchitectureFromSimulink("f14","F14ArchModel",ShowProgress=true)
```

Data Types: logical

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

[inlineComponent](#) | [createSimulinkBehavior](#) | [createStateflowChartBehavior](#) | [linkToModel](#) | [isReference](#) | [Reference Component](#)

### Topics

"Extract Architecture from Simulink Model"

"Implement Component Behavior Using Simulink"

"Decompose and Reuse Components"

"Implement Component Behavior Using Stateflow Charts"

"Create Simulink Subsystem Behavior Using Subsystem Component"



# find

**Package:** systemcomposer.arch

Find architecture model elements using query

## Syntax

```
[paths] = find(model,constraint,Name,Value)
[paths, elements] = find(____)
[elements] = find(____)
[paths] = find(model,constraint,rootArch,Name,Value)
```

## Description

`[paths] = find(model,constraint,Name,Value)` finds all element paths starting from the root architecture of the model that satisfy the constraint query, with additional options specified by one or more name-value arguments.

`[paths, elements] = find(____)` returns the component elements `elements` and their paths that satisfy the constraint query. Follow the syntax above for input arguments. If `rootArch` is not provided, then the function finds model elements in the root architecture of the model. The output argument `paths` contains a fully qualified named path for each component in `elements` from the given root architecture.

`[elements] = find(____)` finds all component, port, or connector elements `elements`, that satisfy the constraint query, with additional options specified by one or more name-value arguments, which must include 'Port' or 'Connector' for 'ElementType'.

`[paths] = find(model,constraint,rootArch,Name,Value)` finds all element paths starting from the specified root architecture `rootArch` that satisfy the constraint query, with additional options specified by one or more name-value arguments.

## Examples

### Find Model Element Paths that Satisfy Query

Import a model and run a query to select architectural elements that have a stereotype based on the specified subconstraint.

```
import systemcomposer.query.*;
scKeylessEntrySystem
modelObj = systemcomposer.openModel("KeylessEntryArchitecture");
find(modelObj,HasStereotype(IsStereotypeDerivedFrom("AutoProfile.BaseComponent")),...
    Recurse=true,IncludeReferenceModels=true)
```

Create a query to find components that contain the letter c in their Name property.

```
constraint = contains(systemcomposer.query.Property("Name"),"c");
find(modelObj,constraint,Recurse=true,IncludeReferenceModels=true)
```

## Find Elements in Architecture Model

Find elements in an architecture model based on a System Composer™ query.

### Create Model

Create an architecture model with two components.

```
m = systemcomposer.createModel("exModel");
comps = m.Architecture.addComponent(["c1", "c2"]);
```

### Create Profile and Stereotypes

Create a profile and stereotypes for your architecture model.

```
pf = systemcomposer.profile.Profile.createProfile("mProfile");
b = pf.addStereotype("BaseComp", AppliesTo="Component", Abstract=true);
s = pf.addStereotype("sComp", Parent=b);
```

### Apply Profile and Stereotypes

Apply the profile and stereotypes to your architecture model.

```
m.Architecture.applyProfile(pf.Name)
comps(1).applyStereotype(s.FullyQualifiedName)
```

### Find the Element

Find the element in your architecture model based on a query.

```
import systemcomposer.query.*
[p, elem] = find(m, HasStereotype(IsStereotypeDerivedFrom("mProfile.BaseComp")), ...
Recurse=true, IncludeReferenceModels=true)
```

```
p = 1x1 cell array
    {'exModel/c1'}
```

```
elem =
    Component with properties:

        IsAdapterComponent: 0
        Architecture: [1x1 systemcomposer.arch.Architecture]
            Name: 'c1'
            Parent: [1x1 systemcomposer.arch.Architecture]
            Ports: [0x0 systemcomposer.arch.ComponentPort]
            OwnedPorts: [0x0 systemcomposer.arch.ComponentPort]
        OwnedArchitecture: [1x1 systemcomposer.arch.Architecture]
            Parameters: [0x0 systemcomposer.arch.Parameter]
            Position: [15 15 65 76]
            Model: [1x1 systemcomposer.arch.Model]
        SimulinkHandle: 2.0005
        SimulinkModelHandle: 4.8828e-04
            UUID: 'e251516a-8174-4786-9703-a95aed4223c5'
            ExternalUID: ''
```

### Clean Up

Remove the model and the profile.

```
cleanUpFindElementsInModel
```

### Find Ports in Architecture Model

Create a model to query and create two components.

```
m = systemcomposer.createModel("exModel");
comps = m.Architecture.addComponent(["c1", "c2"]);
port = comps(1).Architecture.addPort("cport1", "in");
```

Create a query to find ports that contain the letter c in their Name property.

```
constraint = contains(systemcomposer.query.Property("Name"), "c");
find(m, constraint, Recurse=true, IncludeReferenceModels=true, ElementType="Port")
```

### Find Architectural Element Paths That Satisfy Query

```
import systemcomposer.query.*;
scKeylessEntrySystem
modelObj = systemcomposer.openModel("KeylessEntryArchitecture");
find(modelObj, HasStereotype(IsStereotypeDerivedFrom("AutoProfile.BaseComponent")), ...
    modelObj.Architecture, Recurse=true, IncludeReferenceModels=true)
```

## Input Arguments

### model — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

### constraint — Query

query constraint object

Query, specified as a `systemcomposer.query.Constraint` object representing specific conditions.

A constraint can contain a subconstraint that can be joined with another constraint using AND or OR. A constraint can be negated using NOT.

**Query Objects and Conditions for Constraints**

Query Object	Condition
Property	A non-evaluated value for the given property or stereotype property.
PropertyValue	An evaluated property value from a System Composer object or a stereotype property.
HasConnector	A component has a connector that satisfies the given subconstraint.
HasPort	A component has a port that satisfies the given subconstraint.
HasInterface	A port has an interface that satisfies the given subconstraint.
HasInterfaceElement	An interface has an interface element that satisfies the given subconstraint.
HasStereotype	An architecture element has a stereotype that satisfies the given subconstraint.
IsInRange	A property value is within the given range.
AnyComponent	An element is a component and not a port or connector.
IsStereotypeDerivedFrom	A stereotype is derived from the given stereotype.

**rootArch – Root architecture of model**

architecture object | Architecture property of model object

Root architecture of model, specified as a `systemcomposer.arch.Architecture` object or the `Architecture` property of a `systemcomposer.arch.Model` object.

Example: `modelObj.Architecture`

**Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `find(model,constraint,Recurse=true,IncludeReferenceModels=true)`

**Recurse – Option to recursively search model**

true or 1 (default) | false or 0

Option to recursively search model or to only search a specific layer, specified as 1 (`true`) to recursively search or 0 (`false`) to only search the specific layer.

Example: `find(model,constraint,Recurse=true)`

Data Types: `logical`

**IncludeReferenceModels – Option to search for reference architectures**

false or 0 (default) | true or 1

Option to search for reference architectures, specified as a logical.

Example: `find(model,constraint,IncludeReferenceModels=true)`

Data Types: `logical`

### ElementType — Option to search by type

"Component" (default) | "Port" | "Connector"

Option to search by type, specified as one of these types

- "Component" to find components to satisfy the query
- "Port" to find ports to satisfy the query
- "Connector" to find connectors to satisfy the query

Example: `find(model,constraint,ElementType="Port")`

Data Types: `char` | `string`

## Output Arguments

### paths — Element paths

cell array of character vectors

Element paths, returned as a cell array of character vectors that satisfy constraint.

Data Types: `char`

### elements — Elements

element objects

Elements, returned as `systemcomposer.arch.Element` objects that satisfy constraint.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• "Compose Architectures Visually"</li> <li>• "Author Parameters in System Composer Using Parameter Editor"</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019a

### See Also

systemcomposer.query.Constraint | createView | lookup | getQualifiedName | findElementsOfType | findElementsWithStereotype | findElementsWithProperty | findElementsWithInterface

### Topics

“Create Architectural Views Programmatically”



# systemcomposer.profile.Stereotype.find

Find stereotype by name

## Syntax

```
stereotype = systemcomposer.profile.Stereotype.find(name)
```

## Description

`stereotype = systemcomposer.profile.Stereotype.find(name)` finds a stereotype by name.

## Examples

### Find Stereotype

Find a stereotype in the small UAV (unmanned aerial vehicle) model.

```
scExampleSmallUAV
stereotype = systemcomposer.profile.Stereotype.find("UAVComponent.OnboardElement")
```

```
stereotype =
  Stereotype with properties:
      Name: 'OnboardElement'
      Description: 'Represents the base component of UAVComponent'
      Parent: [0x0 systemcomposer.profile.Stereotype]
      AppliesTo: 'Component'
      Abstract: 0
      Icon: 'network'
      ComponentHeaderColor: [210 210 210]
      ConnectorLineColor: [168 168 168]
      ConnectorLineStyle: 'Default'
      FullyQualifiedName: 'UAVComponent.OnboardElement'
      Profile: [1x1 systemcomposer.profile.Profile]
      OwnedProperties: [1x3 systemcomposer.profile.Property]
      Properties: [1x3 systemcomposer.profile.Property]
```

## Input Arguments

### **name** — Name of stereotype

character vector | string

Name of stereotype, specified as a character vector or string in the form "`<profile>.<stereotype>`".

Data Types: char | string

## Output Arguments

### stereotype — Found stereotype

stereotype object

Found stereotype, returned as a `systemcomposer.profile.Stereotype` object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## **Version History**

**Introduced in R2019a**

### **See Also**

[addStereotype](#) | [removeStereotype](#) | [getStereotype](#) | [getDefaultStereotype](#) | [setDefaultStereotype](#)

### **Topics**

["Define Profiles and Stereotypes"](#)

["Use Stereotypes and Profiles"](#)

["Modeling System Architecture of Small UAV"](#)

## systemcomposer.profile.Profile.find

Find profile by name

### Syntax

```
profile = systemcomposer.profile.Profile.find
profile = systemcomposer.profile.Profile.find(profileName)
```

### Description

`profile = systemcomposer.profile.Profile.find` finds all open profiles.

`profile = systemcomposer.profile.Profile.find(profileName)` finds a profile by the specified name, `profileName`.

### Examples

#### Find Profile

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Find the profile by name.

```
profileFound = systemcomposer.profile.Profile.find("LatencyProfile")

profileFound =

  Profile with properties:

      Name: 'LatencyProfile'
  FriendlyName: ''
```

Description: ''  
 Stereotypes: [1x5 systemcomposer.profile.Stereotype]

## Input Arguments

### **profileName** — Name of profile

character vector | string

Name of profile, specified as a character vector or string. Profile must be available on the MATLAB path with a .xml extension.

Example: "LatencyProfile"

Data Types: char | string

## Output Arguments

### **profile** — Found profile

profile object | array of profile objects

Found profile or profiles, returned as a `systemcomposer.profile.Profile` object or an array of `systemcomposer.profile.Profile` objects.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"

Term	Definition	Application	More Information
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

systemcomposer.profile.Profile | open | editor | save | close | closeAll | load | createProfile

### Topics

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

# systemcomposer.allocation.AllocationSet.find

Find loaded allocation set

## Syntax

```
allocSet = systemcomposer.allocation.AllocationSet.find(name)
```

## Description

`allocSet = systemcomposer.allocation.AllocationSet.find(name)` finds a loaded allocation set in the global name space with the given name `name`.

## Examples

### Create Allocation Set and Find the Allocation Set

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Find the allocation set.

```
allocSetFind = systemcomposer.allocation.AllocationSet.find("MyNewAllocation")
```

```
allocSetFind =
```

AllocationSet with properties:

```
    Name: 'MyNewAllocation'
  Description: ''
    Scenarios: [1x1 systemcomposer.allocation.AllocationScenario]
      Dirty: 1
  NeedsRefresh: 0
      UUID: '96e34f0d-fceb-4fb0-872d-2e588308d0e9'
  SourceModel: [1x1 systemcomposer.arch.Model]
  TargetModel: [1x1 systemcomposer.arch.Model]
```

## Input Arguments

### **name** — Name of allocation set

character vector | string

Name of allocation set, specified as a character vector or string.

Example: "MyNewAllocation"

Data Types: char | string

## Output Arguments

### **allocSet** — Allocation set

allocation set object

Allocation set, returned as a `systemcomposer.allocation.AllocationSet` object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

save | load | close | closeAll | synchronizeChanges | getScenario | createScenario | deleteScenario

### Topics

“Create and Manage Allocations Programmatically”



# findElementsOfType

**Package:** systemcomposer.query

Find all elements of specific type

## Syntax

```
elements = findElementsOfType(containerObj, kind)
elements = findElementsOfType(containerObj, kind, Name, Value)
```

## Description

`elements = findElementsOfType(containerObj, kind)` finds all elements of the type `kind`.

`elements = findElementsOfType(containerObj, kind, Name, Value)` finds all elements of a type with additional options specified by one or more name-value arguments.

## Examples

### Find Elements Using Query Programmatic Interfaces

Use functions in the System Composer™ query package to filter elements in a model.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Load the model and the profile.

```
sysModel = systemcomposer.loadModel('mBasicModel');
basicProfile = systemcomposer.loadProfile('mProfile');
```

Find all components in the model.

```
allComps = findElementsOfType(sysModel, "Component")
```

Find all ports in the model.

```
allPorts = findElementsOfType(sysModel, "Port")
```

Find all stereotypes in the model.

```
allStereotypes = findElementsOfType(basicProfile, "Stereotype")
```

Find all interfaces in the model.

```
allInterfaces = findElementsOfType(sysModel, "Interface")
```

Find components in the model with a name containing "c1".

```
nameComponents = findElementsWithProperty(sysModel, "Component", "Name", "c1", "contains")
```

Find all stereotypes in the profile with a name containing "BasePort".

```
basePortStereotype = findElementsWithProperty(basicProfile, "Stereotype", "Name", "BasePort", "eq")
```

Find all components in the model using the stereotype "BasePort".

```
basePorts = findElementsWithStereotype(sysModel, "Port", basePortStereotype)
```

Find all elements using the first two found interfaces.

```
portsUsingInterfaces = findElementsWithInterface(sysModel, "Port", [allInterfaces(1) allInterfaces
```

## Input Arguments

### **containerObj** — Container object

model object | dictionary object | profile object

Container object, specified as one of these options:

- `systemcomposer.arch.Model`
- `systemcomposer.interface.Dictionary`
- `systemcomposer.profile.Profile`

### **kind** — Element type

"Component" | "Port" | "Connector" | "Interface" | "InterfaceElement" | "Stereotype"

Element type, specified as one of these options:

- "Component"
- "Port"
- "Connector"
- "Interface"
- "InterfaceElement"
- "Stereotype"

Data Types: string

### **Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `allComps = systemcomposer.query.findElementsOfType(sysModel, "Component", NegateResult=true, IncludeReferences=false)`

### **NegateResult** — Whether to negate query result

false or 0 (default) | true or 1

Whether to negate query result, specified as a logical.

Data Types: logical

### **IncludeReferences — Option to search through reference architectures**

true or 1 (default) | false or 0

Option to search through reference architectures, specified as a logical.

Data Types: logical

## **Output Arguments**

### **elements — Elements found**

array of component objects | array of port objects | array of connector objects | array of data interface objects | array of data element objects | array of stereotype objects

Elements found, returned as an array of these objects:

- `systemcomposer.arch.Component`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.interface.DataInterface`
- `systemcomposer.interface.DataElement`
- `systemcomposer.profile.Stereotype`

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2023a

### See Also

find | findElementsWithStereotype | findElementsWithProperty | findElementsWithInterface

### Topics

“Create Architectural Views Programmatically”

# findElementsWithStereotype

**Package:** systemcomposer.query

Find all elements with stereotype

## Syntax

```
elements = findElementsWithStereotype(containerObj, kind, stereotypes)
elements = findElementsWithStereotype( ____, Name, Value)
```

## Description

`elements = findElementsWithStereotype(containerObj, kind, stereotypes)` finds all elements of type `kind` with one of these stereotypes specified by `stereotypes` applied.

`elements = findElementsWithStereotype( ____, Name, Value)` finds all elements with stereotypes with additional options specified by one or more name-value arguments.

## Examples

### Find Elements Using Query Programmatic Interfaces

Use functions in the System Composer™ query package to filter elements in a model.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Load the model and the profile.

```
sysModel = systemcomposer.loadModel('mBasicModel');
basicProfile = systemcomposer.loadProfile('mProfile');
```

Find all components in the model.

```
allComps = findElementsOfType(sysModel, "Component")
```

Find all ports in the model.

```
allPorts = findElementsOfType(sysModel, "Port")
```

Find all stereotypes in the model.

```
allStereotypes = findElementsOfType(basicProfile, "Stereotype")
```

Find all interfaces in the model.

```
allInterfaces = findElementsOfType(sysModel, "Interface")
```

Find components in the model with a name containing "c1".

```
nameComponents = findElementsWithProperty(sysModel, "Component", "Name", "c1", "contains")
```

Find all stereotypes in the profile with a name containing "BasePort".

```
basePortStereotype = findElementsWithProperty(basicProfile, "Stereotype", "Name", "BasePort", "eq")
```

Find all components in the model using the stereotype "BasePort".

```
basePorts = findElementsWithStereotype(sysModel, "Port", basePortStereotype)
```

Find all elements using the first two found interfaces.

```
portsUsingInterfaces = findElementsWithInterface(sysModel, "Port", [allInterfaces(1) allInterfaces
```

## Input Arguments

### containerObj — Container object

model object | dictionary object | profile object

Container object, specified as one of these options:

- `systemcomposer.arch.Model`
- `systemcomposer.interface.Dictionary`
- `systemcomposer.profile.Profile`

### kind — Element type

"Component" | "Port" | "Connector" | "Interface"

Element type, specified as one of these options:

- "Component"
- "Port"
- "Connector"
- "Interface"

Data Types: string

### stereotypes — stereotypes

array of stereotype objects

Stereotypes, specified as an array of `systemcomposer.profile.Stereotype` objects.

### Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `basePorts = systemcomposer.query.findElementsWithStereotype(sysModel, "Port", basePortStereotype, NegateResult=true, IncludeReferences=false)`

### NegateResult — Whether to negate query result

false or 0 (default) | true or 1



Whether to negate query result, specified as a logical.

Data Types: `logical`

### **IncludeReferences – Option to search through reference architectures**

`true` or `1` (default) | `false` or `0`

Option to search through reference architectures, specified as a logical.

Data Types: `logical`

## **Output Arguments**

### **elements – Elements found**

array of component objects | array of port objects | array of connector objects | array of data interface objects | array of data element objects | array of stereotype objects

Elements found, returned as an array of these objects:

- `systemcomposer.arch.Component`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.interface.DataInterface`
- `systemcomposer.interface.DataElement`
- `systemcomposer.profile.Stereotype`

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2023a

### See Also

[find](#) | [findElementsOfType](#) | [findElementsWithProperty](#) | [findElementsWithInterface](#)

### Topics

“Create Architectural Views Programmatically”

# findElementsWithProperty

**Package:** systemcomposer.query

Find all elements with property value

## Syntax

```
elements = findElementsWithProperty(containerObj, kind, propertyName,
propertyValue, propertyOperand)
elements = findElementsWithProperty( ____, Name, Value)
```

## Description

`elements = findElementsWithProperty(containerObj, kind, propertyName, propertyValue, propertyOperand)` finds all elements elements of type kind, with the property name propertyName, value propertyValue, or operand propertyOperand.

`elements = findElementsWithProperty( ____, Name, Value)` finds all elements with properties with additional options specified by one or more name-value arguments.

## Examples

### Find Elements Using Query Programmatic Interfaces

Use functions in the System Composer™ query package to filter elements in a model.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Load the model and the profile.

```
sysModel = systemcomposer.loadModel('mBasicModel');
basicProfile = systemcomposer.loadProfile('mProfile');
```

Find all components in the model.

```
allComps = findElementsOfType(sysModel, "Component")
```

Find all ports in the model.

```
allPorts = findElementsOfType(sysModel, "Port")
```

Find all stereotypes in the model.

```
allStereotypes = findElementsOfType(basicProfile, "Stereotype")
```

Find all interfaces in the model.

```
allInterfaces = findElementsOfType(sysModel, "Interface")
```

Find components in the model with a name containing "c1".

```
nameComponents = findElementsWithProperty(sysModel, "Component", "Name", "c1", "contains")
```

Find all stereotypes in the profile with a name containing "BasePort".

```
basePortStereotype = findElementsWithProperty(basicProfile, "Stereotype", "Name", "BasePort", "eq")
```

Find all components in the model using the stereotype "BasePort".

```
basePorts = findElementsWithStereotype(sysModel, "Port", basePortStereotype)
```

Find all elements using the first two found interfaces.

```
portsUsingInterfaces = findElementsWithInterface(sysModel, "Port", [allInterfaces(1) allInterfaces
```

## Input Arguments

### **containerObj** – Container object

model object | dictionary object | profile object

Container object, specified as one of these options:

- `systemcomposer.arch.Model`
- `systemcomposer.interface.Dictionary`
- `systemcomposer.profile.Profile`

### **kind** – Element type

"Component" | "Port" | "Connector" | "Interface" | "InterfaceElement" | "Stereotype"

Element type, specified as one of these options:

- "Component"
- "Port"
- "Connector"
- "Interface"
- "InterfaceElement"
- "Stereotype"

Data Types: `string`

### **propertyName** – Property name

`string`

Property name, specified as a string.

Data Types: `string`

### **propertyValue** – Property value

numeric | `string` | logical | value object

Property value, specified as a numeric, string, logical, or `systemcomposer.query.Value` object.

Data Types: `string` | numeric | logical

**propertyOperand — Property operand**

"eq" | "contains" | "lt" | "gt" | "ge" | "le" | "ne"

Property operand, specified as one of these options:

- eq — Equals
- contains — Contains
- lt — Less than
- gt — Greater than
- ge — Greater than or equal to
- le — Less than or equal to
- ne — Not equal to

Data Types: `string`

**Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

```
Example: basePortStereotype =
systemcomposer.query.findElementsWithProperty(basicProfile,"Stereotype","Name",
"BasePort","eq",NegateResult=true,IncludeReferences=false,UseEvaluatedValue
=false)
```

**NegateResult — Whether to negate query result**

false or 0 (default) | true or 1

Whether to negate query result, specified as a logical.

Data Types: `logical`

**IncludeReferences — Option to search through reference architectures**

true or 1 (default) | false or 0

Option to search through reference architectures, specified as a logical.

Data Types: `logical`

**UseEvaluatedValue — Whether to evaluate value**

true or 1 (default) | false or 0

Whether to evaluate value, specified as a logical.

Data Types: `logical`

**Output Arguments****elements — Elements found**

array of component objects | array of port objects | array of connector objects | array of data interface objects | array of data element objects | array of stereotype objects

Elements found, returned as an array of these objects:

- `systemcomposer.arch.Component`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.interface.DataInterface`
- `systemcomposer.interface.DataElement`
- `systemcomposer.profile.Stereotype`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”



Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2023a

### See Also

find | findElementsOfType | findElementsWithStereotype | findElementsWithInterface

### Topics

“Create Architectural Views Programmatically”

## findElementsWithInterface

**Package:** systemcomposer.query

Find all elements with type set by interface

### Syntax

```
elements = findElementsWithInterface(containerObj,kind,interfaces)
elements = findElementsWithInterface( ____,Name,Value)
```

### Description

`elements = findElementsWithInterface(containerObj,kind,interfaces)` finds all elements of type `kind` whose type is set by one of the specified interfaces `interfaces`.

`elements = findElementsWithInterface( ____,Name,Value)` finds all elements whose type is set by one of the specified interfaces with additional options specified by one or more name-value arguments

### Examples

#### Find Elements Using Query Programmatic Interfaces

Use functions in the System Composer™ query package to filter elements in a model.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Load the model and the profile.

```
sysModel = systemcomposer.loadModel('mBasicModel');
basicProfile = systemcomposer.loadProfile('mProfile');
```

Find all components in the model.

```
allComps = findElementsOfType(sysModel,"Component")
```

Find all ports in the model.

```
allPorts = findElementsOfType(sysModel,"Port")
```

Find all stereotypes in the model.

```
allStereotypes = findElementsOfType(basicProfile,"Stereotype")
```

Find all interfaces in the model.

```
allInterfaces = findElementsOfType(sysModel,"Interface")
```

Find components in the model with a name containing "c1".

```
nameComponents = findElementsWithProperty(sysModel, "Component", "Name", "c1", "contains")
```

Find all stereotypes in the profile with a name containing "BasePort".

```
basePortStereotype = findElementsWithProperty(basicProfile, "Stereotype", "Name", "BasePort", "eq")
```

Find all components in the model using the stereotype "BasePort".

```
basePorts = findElementsWithStereotype(sysModel, "Port", basePortStereotype)
```

Find all elements using the first two found interfaces.

```
portsUsingInterfaces = findElementsWithInterface(sysModel, "Port", [allInterfaces(1) allInterfaces(2)])
```

## Input Arguments

### **containerObj** – Container object

model object | dictionary object

Container object, specified as one of these options:

- `systemcomposer.arch.Model`
- `systemcomposer.interface.Dictionary`

### **kind** – Element type

"Port" | "InterfaceElement"

Element type, specified as one of these options:

- "Port"
- "InterfaceElement"

Data Types: string

### **interfaces** – Interfaces

array of data interface objects

Interfaces, specified as an array of `systemcomposer.interface.DataInterface` objects.

### **Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

```
Example: portsUsingInterfaces =
systemcomposer.query.findElementsWithInterface(sysModel, "Port",
[allInterfaces(1)
allInterfaces(2)], NegateResult=true, IncludeReferences=false)
```

### **NegateResult** – Whether to negate query result

false or 0 (default) | true or 1

Whether to negate query result, specified as a logical.

Data Types: `logical`

### **IncludeReferences – Option to search through reference architectures**

`true` or `1` (default) | `false` or `0`

Option to search through reference architectures, specified as a logical.

Data Types: `logical`

## **Output Arguments**

### **elements – Elements found**

array of component objects | array of port objects | array of connector objects | array of data interface objects | array of data element objects | array of stereotype objects

Elements found, returned as an array of these objects:

- `systemcomposer.arch.Component`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.interface.DataInterface`
- `systemcomposer.interface.DataElement`
- `systemcomposer.profile.Stereotype`

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"



Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2023a

### See Also

find | findElementsOfType | findElementsWithStereotype | findElementsWithProperty

### Topics

“Create Architectural Views Programmatically”

## getActiveChoice

**Package:** systemcomposer.arch

Get active choice on variant component

### Syntax

```
choice = getActiveChoice(variantComponent)
```

### Description

`choice = getActiveChoice(variantComponent)` finds which choice is active for the variant component.

### Examples

#### Get Active Variant Choice

Create a model, get the root architecture, create one variant component, add two choices for the variant component, set the active choice, and get the active choice.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
variant = addVariantComponent(arch, "Component1");
compList = addChoice(variant, ["Choice1", "Choice2"]);
setActiveChoice(variant, compList(2));
comp = getActiveChoice(variant)

comp =
  Component with properties:

    IsAdapterComponent: 0
      Architecture: [1x1 systemcomposer.arch.Architecture]
        Name: 'Choice2'
        Parent: [1x1 systemcomposer.arch.Architecture]
        Ports: [0x0 systemcomposer.arch.ComponentPort]
      OwnedPorts: [0x0 systemcomposer.arch.ComponentPort]
    OwnedArchitecture: [1x1 systemcomposer.arch.Architecture]
      Parameters: [0x0 systemcomposer.arch.Parameter]
      Position: [15 15 65 76]
      Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 218.0010
    SimulinkModelHandle: 0.0012
      UUID: '62caf338-5d9d-48ad-b3c0-d3b0340d598f'
      ExternalUID: ''
```

## Input Arguments

### **variantComponent** — Variant component

variant component object

Variant component, specified as a `systemcomposer.arch.VariantComponent` object.

## Output Arguments

### **choice** — Chosen variant

component object

Chosen variant, returned as a `systemcomposer.arch.Component` object.

## More About

### Definitions

Term	Definition	Application	More Information
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## Version History

Introduced in R2019a

### See Also

`addChoice` | `getChoices` | `setActiveChoice` | Variant Component

### Topics

"Create Variants"

## getAllocatedFrom

**Package:** systemcomposer.allocation

Get allocation source

### Syntax

```
sourceElements = getAllocatedFrom(allocScenario,targetElement)
```

### Description

`sourceElements = getAllocatedFrom(allocScenario,targetElement)` gets all allocated source elements from which a target element `targetElement` is allocated.

### Examples

#### Allocate from Source Component

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario,sourceComp,targetComp);
```

Get allocated from source component allocated to target component.

```
sourceElement = getAllocatedFrom(defaultScenario,targetComp)
```

```
sourceElement =
```

Component with properties:

```
IsAdapterComponent: 0
Architecture: [1x1 systemcomposer.arch.Architecture]
  Name: 'Source_Component'
  Parent: [1x1 systemcomposer.arch.Architecture]
  Ports: [0x0 systemcomposer.arch.ComponentPort]
  OwnedPorts: [0x0 systemcomposer.arch.ComponentPort]
OwnedArchitecture: [1x1 systemcomposer.arch.Architecture]
  Position: [15 15 65 76]
```

```

        Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 2.0001
    SimulinkModelHandle: 1.2207e-04
        UUID: 'c5ab7c89-3ebc-4a19-934b-9b0f473a0737'
    ExternalUID: ''

```

## Input Arguments

### **allocScenario** — Allocation scenario

allocation scenario object

Allocation scenario, specified as a `systemcomposer.allocation.AllocationScenario` object.

### **targetElement** — Target element

element object

Target element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

## Output Arguments

### **sourceElements** — Source elements

array of element objects

Source elements from which specified target element is allocated, returned as an array of `systemcomposer.arch.Element` objects.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>

Term	Definition	Application	More Information
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>• “Establish Traceability Between Architectures and Requirements”</li> <li>• “Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

getAllocatedTo | allocate | deallocate

### Topics

“Create and Manage Allocations Programmatically”

# getAllocatedTo

**Package:** systemcomposer.allocation

Get allocation target

## Syntax

```
targetElements = getAllocatedTo(allocScenario,sourceElement)
```

## Description

`targetElements = getAllocatedTo(allocScenario,sourceElement)` gets all allocated target elements to which the specified source element `sourceElement` is allocated.

## Examples

### Allocate to Target Component

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario,sourceComp,targetComp);
```

Get allocated to target component allocated from source component.

```
targetElement = getAllocatedTo(defaultScenario,sourceComp)
```

```
targetElement =
```

Component with properties:

```
IsAdapterComponent: 0
  Architecture: [1x1 systemcomposer.arch.Architecture]
    Name: 'Target_Component'
    Parent: [1x1 systemcomposer.arch.Architecture]
    Ports: [0x0 systemcomposer.arch.ComponentPort]
    OwnedPorts: [0x0 systemcomposer.arch.ComponentPort]
  OwnedArchitecture: [1x1 systemcomposer.arch.Architecture]
    Position: [15 15 65 76]
```

```

        Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 5.0001
    SimulinkModelHandle: 3.0001
        UUID: '15b4e0ba-f236-4f59-9d30-b46cf170cbda'
    ExternalUID: ''

```

## Input Arguments

### **allocScenario** — Allocation scenario

allocation scenario object

Allocation scenario, specified as a `systemcomposer.allocation.AllocationScenario` object.

### **sourceElement** — Source element

element object

Source element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

## Output Arguments

### **targetElements** — Target elements

array of element objects

Target elements to which source element is allocated, specified as an array of `systemcomposer.arch.Element` objects.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>



Term	Definition	Application	More Information
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>• “Establish Traceability Between Architectures and Requirements”</li> <li>• “Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

allocate | getAllocatedFrom | deallocate

### Topics

“Create and Manage Allocations Programmatically”

## getAllocation

**Package:** systemcomposer.allocation

Get allocation between source and target elements

### Syntax

```
allocation = getAllocation(allocScenario,sourceElement,targetElement)
```

### Description

`allocation = getAllocation(allocScenario,sourceElement,targetElement)` gets the allocation, if one exists, between the source element `sourceElement` and the target element `targetElement`.

### Examples

#### Get Allocation Between Source and Target Components

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario,sourceComp,targetComp);
```

Get the allocation between the source component and the target component.

```
allocation = getAllocation(defaultScenario,sourceComp,targetComp)
```

```
allocation =
```

Allocation with properties:

```
Source: [1x1 systemcomposer.arch.Component]
Target: [1x1 systemcomposer.arch.Component]
```

```
Scenario: [1x1 systemcomposer.allocation.AllocationScenario]
  UUID: 'd83d692d-03fa-4186-977c-ce31b9de9630'
```

## Input Arguments

### **allocScenario — Allocation scenario**

allocation scenario object

Allocation scenario, specified as a `systemcomposer.allocation.AllocationScenario` object.

### **sourceElement — Source element**

element object

Source element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

### **targetElement — Target element**

element object

Target element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

## Output Arguments

### **allocation — Allocation**

allocation object

Allocation between source element and target element, returned as a `systemcomposer.allocation.Allocation` object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>

Term	Definition	Application	More Information
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>• “Establish Traceability Between Architectures and Requirements”</li> <li>• “Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

getAllocatedTo | getAllocatedFrom | deallocate | allocate

### Topics

“Create and Manage Allocations Programmatically”

# getChoices

**Package:** systemcomposer.arch

Get available choices in variant component

## Syntax

```
compList = getChoices(variantComponent)
```

## Description

`compList = getChoices(variantComponent)` returns the list of choices available for a variant component.

## Examples

### Get First Variant Choice

Create a model, get the root architecture, create a one variant component, add two choices for the variant component, and get the first choice.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
variant = addVariantComponent(arch, "Component1");
compList = addChoice(variant, ["Choice1", "Choice2"]);
choices = getChoices(variant);
variantChoice = choices(1)

variantChoice =
  Component with properties:

    IsAdapterComponent: 0
      Architecture: [1x1 systemcomposer.arch.Architecture]
        Name: 'Choice1'
        Parent: [1x1 systemcomposer.arch.Architecture]
        Ports: [0x0 systemcomposer.arch.ComponentPort]
      OwnedPorts: [0x0 systemcomposer.arch.ComponentPort]
    OwnedArchitecture: [1x1 systemcomposer.arch.Architecture]
      Parameters: [0x0 systemcomposer.arch.Parameter]
      Position: [15 15 65 76]
      Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 217.0012
    SimulinkModelHandle: 0.0013
      UUID: 'ddc085f5-07a6-48bc-8db3-b74aca226693'
      ExternalUID: ''
```

## Input Arguments

### **variantComponent** — Variant component

variant component object

Variant component, specified as a `systemcomposer.arch.VariantComponent` object.

## Output Arguments

### **compList** — Choices available for variant component

array of component objects

Choices available for variant component, returned as an array of `systemcomposer.arch.Component` objects.

## More About

### Definitions

Term	Definition	Application	More Information
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## Version History

Introduced in R2019a

### See Also

`addChoice` | `getActiveChoice` | `setActiveChoice` | Variant Component

### Topics

"Create Variants"

# getCondition

**Package:** systemcomposer.arch

Return variant control on choice within variant component

## Syntax

```
expression = getCondition(variantComponent,choice)
```

## Description

`expression = getCondition(variantComponent,choice)` gets the variant control condition for the choice `choice` on the variant component `variantComponent` to choose the active variant choice. If the condition is met on a variant choice, that variant choice becomes the active choice on the variant component.

## Examples

### Get Variant Control Condition

Create a model, get the root architecture, create one variant component, add two choices for the variant component, set a condition on one variant choice to choose the active variant choice, and get the condition.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
mode = 1;
variant = addVariantComponent(arch,"Component1");
compList = addChoice(variant,["Choice1","Choice2"]);
setCondition(variant,compList(2),"mode == 2");
exp = getCondition(variant,compList(2))
```

```
exp =
'mode == 2'
```

## Input Arguments

### **variantComponent** – Variant component

variant component object

Variant component, specified as a `systemcomposer.arch.VariantComponent` object.

### **choice** – Choice in variant component

component object

Choice in variant component, specified as a `systemcomposer.arch.Component` object.

## Output Arguments

### expression — Control string

character vector

Control string that controls the selection of the particular choice, returned as a character vector.

Data Types: char

## More About

### Definitions

Term	Definition	Application	More Information
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## Version History

Introduced in R2019a

### See Also

makeVariant | setActiveChoice | setCondition | addVariantComponent | Variant Component

### Topics

"Create Variants"



# getDefaultElementStereotype

**Package:** systemcomposer.profile

Get default stereotype for elements

## Syntax

```
stereotype = getDefaultElementStereotype(stereotype,elementType)
```

## Description

`stereotype = getDefaultElementStereotype(stereotype,elementType)` gets the default stereotype of the child elements whose parent element of type `elementType` has the stereotype applied.

## Examples

### Get Default Component Stereotype

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Specify the `LatencyProfile.NodeLatency` stereotype as a component stereotype. Set the default component stereotype.

```
nodeLatency.AppliesTo = "Component";
nodeLatency.setDefaultElementStereotype("Component","LatencyProfile.NodeLatency")
```

Get the default component stereotype on `nodeLatency`.

```
stereotype = getDefaultElementStereotype(nodeLatency,"Component")
```

```
stereotype =
```

```
    Stereotype with properties:
```

```

        Name: 'NodeLatency'
    Description: ''
        Parent: [1x1 systemcomposer.profile.Stereotype]
    AppliesTo: 'Component'
    Abstract: 0
        Icon: 'default'
ComponentHeaderColor: [210 210 210]
ConnectorLineColor: [168 168 168]
ConnectorLineStyle: 'Default'
FullyQualifiedName: 'LatencyProfile.NodeLatency'
        Profile: [1x1 systemcomposer.profile.Profile]
    OwnedProperties: [1x1 systemcomposer.profile.Property]
    Properties: [1x3 systemcomposer.profile.Property]

```

## Input Arguments

### elementType — Element type

"Component" | "Port" | "Connector" | "Interface" | "Function"

Element type, specified as "Component", "Port", "Connector", "Interface", or "Function". The element type "Function" is only available for software architectures.

Data Types: char | string

### stereotype — Stereotype

stereotype object

Stereotype, specified as a systemcomposer.profile.Stereotype object.

## Output Arguments

### stereotype — Default stereotype

stereotype object

Default stereotype, returned as a systemcomposer.profile.Stereotype object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	<p>Perform operations on a model:</p> <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> <p>A System Composer model is stored as an SLX file.</p>	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2021b

### See Also

`applyStereotype` | `removeStereotype` | `setDefaultElementStereotype`

### Topics

“Define Profiles and Stereotypes”

# getDefaultStereotype

**Package:** systemcomposer.profile

Get default stereotype for profile

## Syntax

```
stereotype = getDefaultStereotype(profile)
```

## Description

stereotype = getDefaultStereotype(profile) gets the default stereotype for a profile.

## Examples

### Get Default Stereotype

Create a profile for latency characteristics.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfileA");

connLatency = profile.addStereotype("ConnectorLatency",AppliesTo="Connector");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",AppliesTo="Component");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",AppliesTo="Port");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");
```

Set the default stereotype, then get the default stereotype.

```
profile.setDefaultStereotype("NodeLatency");
default = getDefaultStereotype(profile)
```

default =

Stereotype with properties:

```

        Name: 'NodeLatency'
    Description: ''
        Parent: [0x0 systemcomposer.profile.Stereotype]
    AppliesTo: 'Component'
        Abstract: 0
        Icon: ''
ComponentHeaderColor: [210 210 210]
ConnectorLineColor: [168 168 168]
ConnectorLineStyle: 'Default'
FullyQualifiedName: 'LatencyProfileA.NodeLatency'
        Profile: [1x1 systemcomposer.profile.Profile]
```

```
OwnedProperties: [1x1 systemcomposer.profile.Property]
Properties: [1x1 systemcomposer.profile.Property]
```

Close the profile to rerun this example.

```
profile.close(true)
```

## Input Arguments

### **profile** – Profile

profile object

Profile, specified as a `systemcomposer.profile.Profile` object.

## Output Arguments

### **stereotype** – Default stereotype

stereotype object

Default stereotype, returned as a `systemcomposer.profile.Stereotype` object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”

Term	Definition	Application	More Information
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

createProfile | setDefaultStereotype | addStereotype | getStereotype | removeStereotype

### Topics

“Create a Profile and Add Stereotypes”

## getDestinationElement

**Package:** systemcomposer.arch

Gets data elements selected on destination port for connection

### Syntax

```
selectedElems = getDestinationElement(connector)
```

### Description

`selectedElems = getDestinationElement(connector)` gets the selected data elements on a destination port for a connection.

### Examples

#### Get Data Element on Destination Port of Connection

Get the selected element on the destination port for a connection.

Create a model and get its root architecture.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName,true);
rootArch = get(arch,"Architecture");
```

Add a component, create an output port on the component, create an output port on the architecture. and extract both component port objects.

```
newComponent = addComponent(rootArch,"Component2");
outPortComp = addPort(newComponent.Architecture,...
"testSig2","out");
outPortArch = addPort(rootArch,"testSig2","out");
compSrcPort = getPort(newComponent,"testSig2");
archDestPort = getPort(rootArch,"testSig2");
```

Add data interface, create data element, and set the data interface on the architecture port.

```
interface = arch.InterfaceDictionary.addInterface("interface2");
interface.addElement("x");
archDestPort.setInterface(interface);
```

Connect the ports and get the destination element of the connector.

```
conns = connect(compSrcPort,archDestPort,DestinationElement="x");
elem = getDestinationElement(conns)
```

```
elem =
```

```
    1x1 cell array
```



`{'x'}`

## Input Arguments

### **connector** – Connection between ports

connector object

Connection between ports, specified as a `systemcomposer.arch.Connector` object.

## Output Arguments

### **selectedElems** – Selected data element names

character vector

Selected data element names, returned as a character vector.

Data Types: `char`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>

Term	Definition	Application	More Information
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2020b

### See Also

`createModel` | `addPort` | `getPort` | `addComponent` | `addElement` | `addInterface` | `setInterface` | `connect` | `getSourceElement` | `Component`

### Topics

“Specify Source Element or Destination Element for Ports”

# getElement

**Package:** systemcomposer.interface

Get object for element

## Syntax

```
element = getElement(interface,name)
```

## Description

`element = getElement(interface,name)` gets the object for the element with name `name` in the interface specified by `interface`.

## Examples

### Get Object for Named Data Element

Add a data interface `newInterface` to the interface dictionary of the model. Add a data element `newElement` with data type `double`. Then, get the object for the data element.

```
arch = systemcomposer.createModel("newModel",true);
interface = addInterface(arch.InterfaceDictionary,"newInterface");
addElement(interface,"newElement",DataType="double");
element = getElement(interface,"newElement")
```

`element =`

DataElement with properties:

```
Interface: [1x1 systemcomposer.interface.DataInterface]
Name: 'newElement'
Type: [1x1 systemcomposer.ValueType]
UUID: '2d267175-33c2-43a9-be41-albe2774a3cf'
ExternalUUID: ''
```

### Get Object for Named Physical Element

Add a physical interface `newInterface` to the interface dictionary of the model. Add a physical element `newElement` with domain type `electrical.electrical`. Then, get the object for the physical element.

```
arch = systemcomposer.createModel("newModel",true);
interface = addPhysicalInterface(arch.InterfaceDictionary,"newInterface");
addElement(interface,"newElement",Type="electrical.electrical");
element = getElement(interface,"newElement")
```

`element =`

PhysicalElement with properties:

```
Name: 'newElement'
Type: [1x1 systemcomposer.interface.PhysicalDomain]
Interface: [1x1 systemcomposer.interface.PhysicalInterface]
UUID: '25b71628-e904-451a-96ff-f185c5ec60a4'
ExternalUID: ''
```

## Input Arguments

### interface — Interface

data interface object | physical interface object | service interface object

Interface, specified as a `systemcomposer.interface.DataInterface`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### name — Element name

character vector | string

Element name, specified as a character vector or string. An element name must be a valid MATLAB variable name.

Data Types: `char` | `string`

## Output Arguments

### element — Element

data element object | physical element object | function element object

Element, returned as a `systemcomposer.interface.DataElement`, `systemcomposer.interface.PhysicalElement`, or `systemcomposer.interface.FunctionElement` object.

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>“Manage Interfaces with Data Dictionaries”</li> <li>“Reference Data Dictionaries”</li> </ul>

Term	Definition	Application	More Information
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

[addElement](#) | [removeElement](#) | [createDictionary](#) | [getInterfaceNames](#) | [getInterface](#) | [linkDictionary](#) | [getSourceElement](#) | [getDestinationElement](#) | [unlinkDictionary](#)

### Topics

“Specify Physical Interfaces on Ports”  
 “Create Interfaces”  
 “Manage Interfaces with Data Dictionaries”



# getEvaluatedParameterValue

**Package:** systemcomposer.arch

Get evaluated value of parameter from element

## Syntax

```
[value,unit] = getEvaluatedParameterValue(element,paramName)
```

## Description

`[value,unit] = getEvaluatedParameterValue(element,paramName)` gets the evaluated value of the parameter, and, optionally, units `unit` specified on the architectural element, `element`.

## Examples

### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

paramPressure.Type

```
ans =  
  ValueType with properties:  
  
      Name: 'Pressure'  
      DataType: 'double'  
      Dimensions: '[1 1]'  
      Units: 'psi'  
      Complexity: 'real'  
      Minimum: ''  
      Maximum: ''  
      Description: ''  
      Owner: [1x1 systemcomposer.arch.Architecture]  
      Model: [1x1 systemcomposer.arch.Model]  
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'  
      ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
            1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'31'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
            0
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue =
'16'
```

```
paramUnits =
'in'
```

```
isDefault = logical
    1
```

```
paramName =
"Pressure"
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
    1
```

```
paramName =
"Wear"
```

```
paramValue =
'0.25'
```

```
paramUnits =
'in'
```

```
isDefault = logical
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue = 16
```

```
paramUnits =  
'in'  
  
paramName =  
"Pressure"  
  
paramValue = 31  
  
paramUnits =  
'psi'  
  
paramName =  
"Wear"  
  
paramValue = 0.2500  
  
paramUnits =  
'in'
```

Check the evaluated LeftWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"  
  
paramValue = 16  
  
paramUnits =  
'in'  
  
paramName =  
"Pressure"  
  
paramValue = 32  
  
paramUnits =  
'psi'  
  
paramName =  
"Wear"  
  
paramValue = 0.2500  
  
paramUnits =  
'in'
```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")  
  
paramValue =  
'32'  
  
paramUnits =  
'psi'
```

```
isDefault = logical
           1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'34'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
           0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
           1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =  
  Parameter with properties:  
  
    Name: 'Pressure'  
    Value: '30'  
    Type: [1x1 systemcomposer.ValueType]  
    Parent: [1x1 systemcomposer.arch.Component]  
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);  
pressureParam
```

```
pressureParam =  
  Parameter with properties:  
  
    Name: "LeftWheel.Pressure"  
    Value: '32'  
    Type: [1x1 systemcomposer.ValueType]  
    Parent: [1x1 systemcomposer.arch.Architecture]  
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");  
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;  
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce  
  
noiseReduce =  
  Parameter with properties:  
  
    Name: "noiseReduction"
```

```
Value: '30'
Type: [1x1 systemcomposer.ValueType]
Parent: [1x1 systemcomposer.arch.Architecture]
Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the Muffler component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");
save(model)
save(topModel)
```

## Input Arguments

### **element** — Architectural element

architecture object | component object | variant component object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, or `systemcomposer.arch.VariantComponent` object.

### **paramName** — Parameter name

character vector | string

Parameter name, specified as a character vector or string.

Example: "GainArg"

Data Types: char | string

## Output Arguments

### **value** — Parameter value

double (default) | single | int64 | int32 | int16 | int8 | uint64 | uint32 | uint8 | boolean | string | enumeration class name

Parameter value, returned as a data type that depends on how the parameter is defined in the model arguments.

### **unit** — Units of parameter

character vector

Units of parameter, returned as a character vector.

Data Types: char

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”



Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>

Term	Definition	Application	More Information
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022a

### See Also

addParameter | getParameter | resetToDefault | getParameterPromotedFrom | getParameterNames | getParameterValue | setParameterValue | setUnit | resetParameterToDefault

### Topics

“Author Parameters in System Composer Using Parameter Editor”  
 “Access Model Arguments as Parameters on Reference Components”  
 “Use Parameters to Store Instance Values with Components”

# getEvaluatedPropertyValue

**Package:** systemcomposer.arch

Get evaluated value of property from element

## Syntax

```
value = getEvaluatedPropertyValue(element,property)
```

## Description

`value = getEvaluatedPropertyValue(element,property)` gets the evaluated value of a property specified on the architectural element.

## Examples

### Get Evaluated Property Value

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

Create a model with a component.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
comp = addComponent(arch,"Component");
```

Apply the profile to the model and apply the stereotype to the component.

```
model.applyProfile("LatencyProfile");
comp.applyStereotype("LatencyProfile.electricalComponent");
```

Get the property value

```
value = getEvaluatedPropertyValue(comp,"LatencyProfile.electricalComponent.latency")
```

```
value =
```

```
10
```

## Input Arguments

### element — Architectural element

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

**property – Property name**

`character vector` | `string`

Property name, specified as a character vector or string in the form "`<profile>.<stereotype>.<property>`".

Data Types: `char` | `string`

## Output Arguments

**value – Property value**

`double` (default) | `single` | `int64` | `int32` | `int16` | `int8` | `uint64` | `uint32` | `uint8` | `boolean` | `string` | `enumeration class name`

Property value, returned as a data type that depends on how the property is defined in the profile.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

setProperty | getProperty | getStereotypeProperties | getPropertyValue

### Topics

“Write Analysis Function”

# getFunctionArgument

**Package:** systemcomposer.interface

Get function argument on function element

## Syntax

```
arg = getFunctionArgument(functionElem, argName)
```

## Description

`arg = getFunctionArgument(functionElem, argName)` gets the function argument `argName` specified by a function prototype from a function defined by the function element `functionElem`.

## Examples

### Get Function Argument

Create a new model.

```
model = systemcomposer.createModel("archModel", "SoftwareArchitecture", true);
```

Create a service interface.

```
interface = addServiceInterface(model.InterfaceDictionary, "newServiceInterface");
```

Create a function element.

```
element = addElement(interface, "newFunctionElement");
```

Set a function prototype to add function arguments.

```
setFunctionPrototype(element, "y=f0(u)")
```

Get a function argument.

```
argument = getFunctionArgument(element, "y")
```

```
argument =
```

FunctionArgument with properties:

```
Interface: [1x1 systemcomposer.interface.ServiceInterface]
Element: [1x1 systemcomposer.interface.FunctionElement]
Name: 'y'
Type: [1x1 systemcomposer.ValueType]
Dimensions: '1'
Description: ''
```

```

        UUID: '018b4e55-fa8f-4250-ac2b-df72bf620feb'
    ExternalUID: ''

```

## Input Arguments

### **functionElem** – Function element

function element object

Function element, specified as a `systemcomposer.interface.FunctionElement` object.

### **argName** – Argument name

character vector | string

Argument name, specified as a character vector or string.

Example: "y"

Data Types: `char` | `string`

## Output Arguments

### **arg** – Function argument

function argument object

Function argument, returned as a `systemcomposer.interface.FunctionArgument` object.

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Create Software Architecture from Component”</li> </ul>



Term	Definition	Application	More Information
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2022a

### See Also

`addElement` | `createDictionary` | `addServiceInterface` | `getInterface` | `getInterfaceNames` | `removeInterface` | `linkDictionary` | `Adapter` | `addValueType` | `setFunctionPrototype` | `setAsynchronous`

### Topics

“Author Service Interfaces for Client-Server Communication”

“Client-Server Interfaces in Class Diagram View”

“Define Port Interfaces Between Components”

# getInterface

**Package:** systemcomposer.interface

Get object for named interface in interface dictionary

## Syntax

```
interface = getInterface(dictionary,name)
interface = getInterface(dictionary,name,Name,Value)
```

## Description

`interface = getInterface(dictionary,name)` gets the object for a named interface in the interface dictionary.

`interface = getInterface(dictionary,name,Name,Value)` gets the object for a named interface in the interface dictionary with additional options.

## Examples

### Add Data Interface and Get Data Interface

Add a data interface `newInterface` to the interface dictionary of the model. Obtain the data interface object. Confirm by opening the **Interface Editor**.

```
arch = systemcomposer.createModel("newModel",true);
addInterface(arch.InterfaceDictionary,"newInterface");
interface = getInterface(arch.InterfaceDictionary,"newInterface")
```

```
interface =
```

```
DataInterface with properties:
```

```
    Owner: [1x1 systemcomposer.interface.Dictionary]
      Name: 'newInterface'
  Elements: [0x0 systemcomposer.interface.DataElement]
    Model: [1x1 systemcomposer.arch.Model]
      UUID: '205cdd2f-12bc-4bbb-a4a7-75d0ab18adb8'
  ExternalUID: ''
```

### Add Physical Interface and Get Physical Interface

Add a physical interface `newInterface` to the interface dictionary of the model. Obtain the physical interface object. Confirm by opening the **Interface Editor**.

```
arch = systemcomposer.createModel("newModel",true);
addPhysicalInterface(arch.InterfaceDictionary,"newInterface");
interface = getInterface(arch.InterfaceDictionary,"newInterface")
```

```
interface =
    PhysicalInterface with properties:
        Owner: [1x1 systemcomposer.interface.Dictionary]
        Name: 'newInterface'
        Elements: [0x0 systemcomposer.interface.PhysicalElement]
        Model: [1x1 systemcomposer.arch.Model]
        UUID: '6110207d-2d6d-470e-9bf5-f0e6f6914685'
        ExternalUUID: ''
```

## Input Arguments

### dictionary — Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

### name — Name of interface

character vector | string

Name of interface, specified as a character vector or string.

Example: "newInterface"

Data Types: char | string

### Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example: `getInterface(dict,"newInterface",ReferenceDictionary="")`

### ReferenceDictionary — Reference dictionary

character vector | string

Reference dictionary to search for interfaces, specified as a character vector or string with the `.sldd` extension. Enter an empty character vector or string to include all referenced dictionaries in the search.

Example:

```
getInterface(dict,"newInterface",ReferenceDictionary="referenceDictionary.sldd")
```

Data Types: char | string

## Output Arguments

### interface – Interface

data interface object | physical interface object | service interface object | value type object

Interface, returned as a `systemcomposer.interface.DataInterface`, `systemcomposer.interface.PhysicalInterface`, `systemcomposer.interface.ServiceInterface`, or `systemcomposer.ValueType` object.

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	"Create Value Types as Interfaces"
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	"Define Owned Interfaces Local to Ports"
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the "Interface Adapter" dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• "Interface Adapter"</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a



**See Also**

[addElement](#) | [getInterfaceNames](#) | [removeElement](#) | [addInterface](#) | [addValueType](#) | [addPhysicalInterface](#) | [addServiceInterface](#) | [Adapter](#)

**Topics**

[“Specify Physical Interfaces on Ports”](#)

[“Create Interfaces”](#)

[“Manage Interfaces with Data Dictionaries”](#)

## getInterfaceNames

**Package:** systemcomposer.interface

Get names of all interfaces in interface dictionary

### Syntax

```
interfaceNames = getInterfaceNames(dictionary)
```

### Description

`interfaceNames = getInterfaceNames(dictionary)` gets the names of all interfaces in the interface dictionary.

### Examples

#### Get Interface Names

Create a model, add three data interfaces, and get interface names. Confirm by opening the **Interface Editor**.

```
arch = systemcomposer.createModel("newModel", true);
addInterface(arch.InterfaceDictionary, "newInterfaceA");
addInterface(arch.InterfaceDictionary, "newInterfaceB");
addInterface(arch.InterfaceDictionary, "newInterfaceC");
interfaceNames = getInterfaceNames(arch.InterfaceDictionary)

interfaceNames =

    1×3 cell array

    {'newInterfaceA'}    {'newInterfaceB'}    {'newInterfaceC'}
```

### Input Arguments

#### **dictionary** — Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

### Output Arguments

#### **interfaceNames** — Interface names

cell array of character vectors

Interface names, returned as a cell array of character vectors.

Data Types: char

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	"Create Value Types as Interfaces"
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	"Define Owned Interfaces Local to Ports"
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the "Interface Adapter" dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• "Interface Adapter"</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

**See Also**

[addInterface](#) | [getInterface](#) | [removeInterface](#) | [addValueType](#) | [addServiceInterface](#) | [addPhysicalInterface](#) | [Adapter](#)

**Topics**

[“Specify Physical Interfaces on Ports”](#)

[“Create Interfaces”](#)

[“Manage Interfaces with Data Dictionaries”](#)

## getParameter

**Package:** systemcomposer.arch

Get parameter from architecture or component

### Syntax

```
param = getParameter(arch,paramName)
```

### Description

`param = getParameter(arch,paramName)` gets a parameter, `param`, with the name `paramName` from the architecture `arch`.

### Examples

#### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

**paramPressure.Type**

```
ans =
  ValueType with properties:
      Name: 'Pressure'
      DataType: 'double'
      Dimensions: '[1 1]'
      Units: 'psi'
      Complexity: 'real'
      Minimum: ''
      Maximum: ''
      Description: ''
      Owner: [1x1 systemcomposer.arch.Architecture]
      Model: [1x1 systemcomposer.arch.Model]
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'
      ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue =
'16'
```

```
paramUnits =
'in'
```

```
isDefault = logical
    1
```

```
paramName =
"Pressure"
```

```
paramValue =
'31'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
    0
```

```
paramName =
"Wear"
```

```
paramValue =
'0.25'
```

```
paramUnits =
'in'
```

```
isDefault = logical  
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue = 16
```



```

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 31

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Check the evaluated LeftWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

paramValue = 16

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 32

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```

[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

```

```
isDefault = logical
  1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'34'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =
  Parameter with properties:
    Name: 'Pressure'
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Component]
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '32'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce
noiseReduce =
  Parameter with properties:
    Name: "noiseReduction"
```

```
Value: '30'  
Type: [1x1 systemcomposer.ValueType]  
Parent: [1x1 systemcomposer.arch.Architecture]  
Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the Muffler component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");  
save(model)  
save(topModel)
```

## Input Arguments

### **arch** — Architecture

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **paramName** — Parameter name

character vector | string

Parameter name, specified as a character vector or string.

Example: "GainArg"

Data Types: `char` | `string`

## Output Arguments

### **param** — Parameter

parameter object

Parameter, returned as a `systemcomposer.arch.Parameter` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	<p>Perform operations on a model:</p> <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> <p>A System Composer model is stored as an SLX file.</p>	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• "Implement Component Behavior Using Simulink"</li> <li>• "Create Architecture Reference"</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• "Author Parameters in System Composer Using Parameter Editor"</li> <li>• "Access Model Arguments as Parameters on Reference Components"</li> <li>• "Use Parameters to Store Instance Values with Components"</li> </ul>

Term	Definition	Application	More Information
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022b

### See Also

addParameter | resetToDefault | getParameterPromotedFrom |  
 getEvaluatedParameterValue | getParameterValue | setParameterValue | setUnit |  
 getParameterNames | resetParameterToDefault

### Topics

“Author Parameters in System Composer Using Parameter Editor”  
 “Access Model Arguments as Parameters on Reference Components”  
 “Use Parameters to Store Instance Values with Components”

# getParameterDefinition

**Package:** `systemcomposer.arch`

(Not recommended) Get instance-specific parameter definition

---

**Note** The `getParameterDefinition` function is not recommended. Use the `systemcomposer.arch.Parameter` object with its `Type` property instead. For more information, see “Compatibility Considerations”.

---

## Syntax

```
paramDef = getParameterDefinition(arch,paramName)
```

## Description

`paramDef = getParameterDefinition(arch,paramName)` gets the instance-specific parameter definition object for a given architecture, `arch`, and parameter name, `paramName`.

## Input Arguments

### **arch — Architecture**

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **paramName — Parameter name**

character vector | string

Parameter name, specified as a character vector or string.

Example: "GainArg"

Data Types: `char` | `string`

## Output Arguments

### **paramDef — Parameter definition**

parameter definition object

Parameter definition, returned as a `systemcomposer.parameter.ParameterDefinition` object.

## Version History

**Introduced in R2022a**

**R2022b\_plus: `getParameterDefinition` function is not recommended**

*Not recommended starting in R2022b\_plus*



The `getParameterDefinition` function is not recommended. Use the `systemcomposer.arch.Parameter` object with its `Type` property instead.

### **See Also**

`getEvaluatedParameterValue` | `getParameterNames` | `getParameterValue` | `setParameterValue` | `setUnit` | `resetParameterToDefault`

### **Topics**

“Access Model Arguments as Parameters on Reference Components”

“Use Parameters to Store Instance Values with Components”

## getParameterNames

**Package:** systemcomposer.arch

Get parameter names on element

### Syntax

```
paramNames = getParameterNames(element)
```

### Description

`paramNames = getParameterNames(element)` gets the names of the parameters available on the specified architectural element, `element`.

### Examples

#### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the Pressure parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the Pressure parameter.

**paramPressure.Type**

```
ans =
  ValueType with properties:
      Name: 'Pressure'
      DataType: 'double'
      Dimensions: '[1 1]'
      Units: 'psi'
      Complexity: 'real'
      Minimum: ''
      Maximum: ''
      Description: ''
      Owner: [1x1 systemcomposer.arch.Architecture]
      Model: [1x1 systemcomposer.arch.Model]
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'
      ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue =
'16'
```

```
paramUnits =
'in'
```

```
isDefault = logical
    1
```

```
paramName =
"Pressure"
```

```
paramValue =
'31'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
    0
```

```
paramName =
"Wear"
```

```
paramValue =
'0.25'
```

```
paramUnits =
'in'
```

```
isDefault = logical  
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue = 16
```

```

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 31

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Check the evaluated LeftWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

paramValue = 16

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 32

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```

[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

```

```
isDefault = logical  
  1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")  
[paramValue, paramUnits, isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =  
'34'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
  0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue, paramUnits, isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
  1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture, Path="mAxleArch/LeftWheel", Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";  
pressureParam
```

```
pressureParam =  
  Parameter with properties:  
  
    Name: "LeftWheel.Pressure"  
    Value: '30'  
    Type: [1x1 systemcomposer.ValueType]  
    Parent: [1x1 systemcomposer.arch.Architecture]  
    Unit: 'psi'
```

Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =
  Parameter with properties:
    Name: 'Pressure'
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Component]
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '32'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce
noiseReduce =
  Parameter with properties:
    Name: "noiseReduction"
```

```
Value: '30'
Type: [1x1 systemcomposer.ValueType]
Parent: [1x1 systemcomposer.arch.Architecture]
Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the Muffler component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");
save(model)
save(topModel)
```

## Input Arguments

### **element** — Architectural element

architecture object | component object | variant component object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, or `systemcomposer.arch.VariantComponent` object.

## Output Arguments

### **paramNames** — Parameter names

array of strings

Parameter names, returned as an array of strings.

Data Types: `string`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>



Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022a

### See Also

addParameter | getParameter | resetToDefault | getParameterPromotedFrom |  
getEvaluatedParameterValue | getParameterValue | setParameterValue | setUnit |  
resetParameterToDefault

### Topics

“Author Parameters in System Composer Using Parameter Editor”  
“Access Model Arguments as Parameters on Reference Components”  
“Use Parameters to Store Instance Values with Components”

## getParameterPromotedFrom

**Package:** systemcomposer.arch

Get source parameter promoted from

### Syntax

```
source = getParameterPromotedFrom(param)
```

### Description

`source = getParameterPromotedFrom(param)` gets the source parameter source that the given parameter `param` is promoted from.

### Examples

#### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

**paramPressure.Type**

```
ans =
  ValueType with properties:
      Name: 'Pressure'
      DataType: 'double'
      Dimensions: '[1 1]'
      Units: 'psi'
      Complexity: 'real'
      Minimum: ''
      Maximum: ''
      Description: ''
      Owner: [1x1 systemcomposer.arch.Architecture]
      Model: [1x1 systemcomposer.arch.Model]
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'
      ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue =
'16'
```

```
paramUnits =
'in'
```

```
isDefault = logical
1
```

```
paramName =
"Pressure"
```

```
paramValue =
'31'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
0
```

```
paramName =
"Wear"
```

```
paramValue =
'0.25'
```

```
paramUnits =
'in'
```

```
isDefault = logical  
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue = 16
```

```

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 31

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Check the evaluated LeftWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

paramValue = 16

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 32

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```

[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

```

```
isDefault = logical
  1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'34'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";
pressureParam
```

```
pressureParam =
  Parameter with properties:
```

```
    Name: "LeftWheel.Pressure"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```



Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =
  Parameter with properties:
    Name: 'Pressure'
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Component]
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '32'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce
noiseReduce =
  Parameter with properties:
    Name: "noiseReduction"
```

```
Value: '30'
Type: [1x1 systemcomposer.ValueType]
Parent: [1x1 systemcomposer.arch.Architecture]
Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the Muffler component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");
save(model)
save(topModel)
```

## Input Arguments

### **param** — Parameter

parameter object

Parameter, specified as a `systemcomposer.arch.Parameter` object.

## Output Arguments

### **source** — Source parameter

parameter object

Source parameter, returned as a `systemcomposer.arch.Parameter` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022b

### See Also

addParameter | getParameter | resetToDefault | getEvaluatedParameterValue |  
getParameterNames | setParameterValue | resetParameterToDefault |  
getParameterValue | setUnit

### Topics

“Author Parameters in System Composer Using Parameter Editor”  
“Access Model Arguments as Parameters on Reference Components”  
“Use Parameters to Store Instance Values with Components”

# getParameterValue

**Package:** systemcomposer.arch

Get value of parameter

## Syntax

```
[value,unit,isDefault] = getParameterValue(element,paramName)
```

## Description

`[value,unit,isDefault] = getParameterValue(element,paramName)` gets the non-evaluated parameter value of the parameter specified by `paramName` for the provided architectural element, `element`.

## Examples

### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

**paramPressure.Type**

```
ans =
  ValueType with properties:
      Name: 'Pressure'
      DataType: 'double'
      Dimensions: '[1 1]'
      Units: 'psi'
      Complexity: 'real'
      Minimum: ''
      Maximum: ''
      Description: ''
      Owner: [1x1 systemcomposer.arch.Architecture]
      Model: [1x1 systemcomposer.arch.Model]
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'
      ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue =
'16'
```

```
paramUnits =
'in'
```

```
isDefault = logical
           1
```

```
paramName =
"Pressure"
```

```
paramValue =
'31'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
           0
```

```
paramName =
"Wear"
```

```
paramValue =
'0.25'
```

```
paramUnits =
'in'
```

```
isDefault = logical  
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue = 16
```



```

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 31

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Check the evaluated LeftWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

paramValue = 16

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 32

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```

[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

```

```
isDefault = logical
  1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'34'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =
  Parameter with properties:
    Name: 'Pressure'
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Component]
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '32'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce
noiseReduce =
  Parameter with properties:
    Name: "noiseReduction"
```

```
Value: '30'  
Type: [1x1 systemcomposer.ValueType]  
Parent: [1x1 systemcomposer.arch.Architecture]  
Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the `Muffler` component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");  
save(model)  
save(topModel)
```

## Input Arguments

### **element** — Architectural element

architecture object | component object | variant component object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, or `systemcomposer.arch.VariantComponent` object.

### **paramName** — Parameter name

character vector | string

Parameter name, specified as a character vector or string.

Example: "GainArg"

Data Types: char | string

## Output Arguments

### **value** — Parameter value

character vector

Parameter value, returned as a character vector.

Data Types: char

### **unit** — Units of parameter

character vector

Units of parameter, returned as a character vector.

Data Types: char

### **isDefault** — Whether parameter value is default

true or 1 | false or 0

Whether parameter value is default, returned as a logical.

Data Types: logical

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>

Term	Definition	Application	More Information
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>“Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>“Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022a

### See Also

addParameter | getParameter | resetToDefault | getParameterPromotedFrom | getEvaluatedParameterValue | getParameterNames | setParameterValue | setUnit | resetParameterToDefault

### Topics

“Author Parameters in System Composer Using Parameter Editor”  
 “Access Model Arguments as Parameters on Reference Components”  
 “Use Parameters to Store Instance Values with Components”

## getPort

**Package:** systemcomposer.arch

Get port from component

### Syntax

```
port = getPort(compObj, portName)
```

### Description

`port = getPort(compObj, portName)` gets the port on the component `compObj` with a specified name `portName`.

### Examples

#### Connect Ports

Create and connect two ports in System Composer.

Create a top-level architecture model.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName, true);  
rootArch = get(arch, "Architecture");
```

Create two new components.

```
names = ["Component1", "Component2"];  
newComponents = addComponent(rootArch, names);
```

Add ports to the components.

```
outPort1 = addPort(newComponents(1).Architecture, "testSig", "out");  
inPort1 = addPort(newComponents(2).Architecture, "testSig", "in");
```

Extract the component ports.

```
srcPort = getPort(newComponents(1), "testSig");  
destPort = getPort(newComponents(2), "testSig");
```

Connect the ports.

```
conns = connect(srcPort, destPort);
```

View the model.

```
systemcomposer.openModel(modelName);
```

Improve the model layout.



```
Simulink.BlockDiagram.arrangeSystem(modelName)
```

## Input Arguments

### **compObj — Component**

component object

Component to get port from, specified as a `systemcomposer.arch.Component` or `systemcomposer.arch.VariantComponent` object.

### **portName — Name of port**

character vector | string

Name of port, specified as a character vector or string.

Example: "testSig"

Data Types: char | string

## Output Arguments

### **port — Component port**

component port

Component port, returned as a `systemcomposer.arch.ComponentPort` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## **Version History**

**Introduced in R2019a**

### **See Also**

[createModel](#) | [addPort](#) | [addComponent](#) | [connect](#) | [Component](#)

## getProperty

**Package:** systemcomposer.arch

Get property value corresponding to stereotype applied to element

### Syntax

```
[propertyValue,propertyUnits] = getProperty(element,propertyName)
```

### Description

[propertyValue,propertyUnits] = getProperty(element,propertyName) obtains the value and units of the property specified in the propertyName argument. Get the property corresponding to an applied stereotype by qualified name "<profile>.<stereotype>.<property>".

### Examples

#### Get Property from Component

Get the weight property from a component with sysComponent stereotype applied.

Create a model with a component called Component.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
comp = addComponent(arch,"Component");
```

Create a profile with a stereotype with a property, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("sysProfile");
base = profile.addStereotype("sysComponent");
base.addProperty("weight",Type="double",DefaultValue="10",Units="g");
model.applyProfile("sysProfile");
```

Apply the stereotype to the component, and set a new weight property.

```
applyStereotype(comp,"sysProfile.sysComponent")
setProperty(comp,"sysProfile.sysComponent.weight","5","g")
```

Get the weight property with units.

```
[val,units] = getProperty(comp,"sysProfile.sysComponent.weight")
```

```
val =
```

```
    '5'
```

```
units =
```

'g'

## Input Arguments

### **element — Architectural element**

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### **propertyName — Name of property**

character vector | string

Name of property, specified as a character vector or string in the form "`<profile>.<stereotype>.<property>`".

Data Types: char | string

## Output Arguments

### **propertyValue — Value of property**

character vector

Value of property, returned as a character vector.

Data Types: char

### **propertyUnits — Units of property**

character vector

Units of property to interpret property values, returned as a character vector.

Data Types: char

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"

Term	Definition	Application	More Information
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	“Define Physical Ports on Component”
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	“Architecture Model with Simscape Behavior for a DC Motor”



Term	Definition	Application	More Information
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on Ports”
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	“Describe Component Behavior Using Simscape”

## Version History

Introduced in R2019a

### See Also

[setProperty](#) | [removeProperty](#) | [addProperty](#) | [getStereotypeProperties](#)

### Topics

“Set Properties for Analysis”

## getPropertyValue

**Package:** systemcomposer.arch

Get value of architecture property

### Syntax

```
value = getPropertyValue(element,property)
```

### Description

`value = getPropertyValue(element,property)` gets the non-evaluated property value for the provided architectural element.

### Examples

#### Get Property Value

Create a profile, add a component stereotype, and add a property with a default value.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
stereotype = addStereotype(profile,"electricalComponent",AppliesTo="Component");
stereotype.addProperty("latency",Type="double",DefaultValue="10");
```

Create a model with a component.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
comp = addComponent(arch,"Component");
```

Apply the profile to the model and apply the stereotype to the component. Open the **Profile Editor**.

```
model.applyProfile("LatencyProfile")
comp.applyStereotype("LatencyProfile.electricalComponent")
systemcomposer.profile.editor(profile)
```

Get the property value.

```
value = getPropertyValue(comp,"LatencyProfile.electricalComponent.latency")
value =
    '10'
```

### Input Arguments

#### element — Architectural element

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`,

systemcomposer.arch.ComponentPort, systemcomposer.arch.ArchitecturePort, systemcomposer.arch.Connector, systemcomposer.arch.PhysicalConnector, systemcomposer.arch.Function, systemcomposer.interface.DataInterface, systemcomposer.ValueType, systemcomposer.interface.PhysicalInterface, or systemcomposer.interface.ServiceInterface object.

### property – Property name

character vector | string

Property name, specified as a character vector or string in the form "<profile>.<stereotype>.<property>".

Data Types: char | string

## Output Arguments

### value – Property value

character vector

Property value, returned as a character vector.

Data Types: char

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”

Term	Definition	Application	More Information
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2019a

### See Also

`setProperty` | `getStereotypeProperties` | `getProperty` | `getEvaluatedPropertyValue`

### Topics

"Write Analysis Function"

# getScenario

**Package:** systemcomposer.allocation

Get allocation scenario

## Syntax

```
scenario = getScenario(allocSet,name)
```

## Description

`scenario = getScenario(allocSet,name)` gets the allocation scenario in the allocation set `allocSet` with the given name `name`, if one exists.

## Examples

### Create Allocation Set and Get Default Scenario

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1")
```

```
defaultScenario =
```

AllocationScenario with properties:

```
    Name: 'Scenario 1'
  Description: ''
 AllocationSet: [1x1 systemcomposer.allocation.AllocationSet]
  Allocations: [0x0 systemcomposer.allocation.Allocation]
    UUID: '6cde23e8-7c72-4fa0-8f51-e65290208564'
```

## Input Arguments

### **allocSet** — Allocation set

allocation set object

Allocation set, specified as a `systemcomposer.allocation.AllocationSet` object.

### **name** — Name of allocation scenario

character vector | string

Name of allocation scenario, specified as a character vector or string.

Example: "Scenario 1"

Data Types: char | string

## Output Arguments

### scenario — Allocation scenario

allocation scenario object

Allocation scenario, returned as a `systemcomposer.allocation.AllocationScenario` object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <code>Scenario 1</code> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as <code>MLDATX</code> files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

`createScenario` | `deleteScenario` | `close` | `load` | `save` | `synchronizeChanges` | `find` | `closeAll`

### Topics

“Create and Manage Allocations Programmatically”



# getSourceElement

**Package:** systemcomposer.arch

Gets data elements selected on source port for connection

## Syntax

```
selectedElems = getSourceElement(connector)
```

## Description

`selectedElems = getSourceElement(connector)` gets the selected data elements on a source port for a connection.

## Examples

### Get Data Element on Source Port of Connection

Get the selected data element on the source port for a connection.

Create a model and get its root architecture.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName,true);
rootArch = get(arch,"Architecture");
```

Add a component, create an input port on the component, create an input port on the architecture. and extract both component port objects.

```
newComponent = addComponent(rootArch,"Component1");
inPortComp = addPort(newComponent.Architecture,...
"testSig1","in");
inPortArch = addPort(rootArch,"testSig1","in");
compDestPort = getPort(newComponent,"testSig1");
archSrcPort = getPort(rootArch,"testSig1");
```

Add data interface, create data element, and set the data interface on the architecture port.

```
interface = arch.InterfaceDictionary.addInterface("interface1");
interface.addElement("x");
archSrcPort.setInterface(interface);
```

Connect the ports and get the source element of the connector.

```
conns = connect(archSrcPort,compDestPort,SourceElement="x");
elem = getSourceElement(conns)
```

```
elem =
```

```
  1x1 cell array
```

{'x'}

## Input Arguments

### connector — Connection between ports

connector object

Connection between ports, specified as a `systemcomposer.arch.Connector` object.

## Output Arguments

### selectedElems — Selected data element names

character vector

Selected data element names, returned as a character vector.

Data Types: `char`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>

Term	Definition	Application	More Information
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2020b

### See Also

`createModel` | `addPort` | `getPort` | `addComponent` | `addElement` | `addInterface` | `setInterface` | `connect` | `getDestinationElement` | `Component`

### Topics

“Specify Source Element or Destination Element for Ports”

## getStereotype

**Package:** systemcomposer.profile

Find stereotype in profile by name

### Syntax

```
stereotype = getStereotype(profile,name)
```

### Description

stereotype = getStereotype(profile,name) finds a stereotype in a profile by name.

### Examples

#### Get Stereotype by Name

Create a profile for latency characteristics.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfileB");

connLatency = profile.addStereotype("ConnectorLatency",AppliesTo="Connector");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",AppliesTo="Component");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",AppliesTo="Port");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");
```

Get the stereotype ConnectorLatency in the profile.

```
stereotype = getStereotype(profile,"ConnectorLatency")
```

```
stereotype =
```

```
  Stereotype with properties:
```

```
      Name: 'ConnectorLatency'
  Description: ''
      Parent: [0x0 systemcomposer.profile.Stereotype]
  AppliesTo: 'Connector'
    Abstract: 0
      Icon: ''
ComponentHeaderColor: [210 210 210]
ConnectorLineColor: [168 168 168]
ConnectorLineStyle: 'Default'
FullyQualifiedName: 'LatencyProfileB.ConnectorLatency'
      Profile: [1x1 systemcomposer.profile.Profile]
    OwnedProperties: [1x2 systemcomposer.profile.Property]
```

Properties: [1x2 systemcomposer.profile.Property]

Close the profile to rerun this example.

```
profile.close(true)
```

## Input Arguments

### **profile** — Profile

profile object

Profile, specified as a `systemcomposer.profile.Profile` object.

### **name** — Stereotype name

character vector | string

Stereotype name, specified as a character vector or string. The name of the stereotype must be unique within the profile.

Data Types: `char` | `string`

## Output Arguments

### **stereotype** — Stereotype

stereotype object

Stereotype found, returned as a `systemcomposer.profile.Stereotype` object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

`addStereotype` | `removeStereotype` | `getDefaultStereotype` | `setDefaultStereotype`



**Topics**

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

# getStereotypeProperties

**Package:** systemcomposer.arch

Get stereotype property names on element

## Syntax

```
propNames = getStereotypeProperties(archElement)
```

## Description

`propNames = getStereotypeProperties(archElement)` returns an array of stereotype property names on the specified architecture of an element.

## Examples

### Get Stereotype Properties

Create a profile, add a component stereotype, and add properties with default values.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
stereotype = addStereotype(profile,"electricalComponent",AppliesTo="Component");
stereotype.addProperty("latency",Type="double",DefaultValue="10");
stereotype.addProperty("mass",Type="double",DefaultValue="20");
```

Create a model with a component.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
comp = addComponent(arch,"Component");
```

Apply the profile to the model and apply the stereotype to the component. Open the **Profile Editor**.

```
model.applyProfile("LatencyProfile");
comp.applyStereotype("LatencyProfile.electricalComponent");
systemcomposer.profile.editor(profile)
```

Get stereotype properties on the architecture of the component.

```
properties = getStereotypeProperties(comp.Architecture)

properties =
    1x2 string array
    "LatencyProfile.electricalComponent.latency"    "LatencyProfile.electricalComponent.mass"
```

## Input Arguments

### archElement — Model element architecture

architecture object | architecture port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Model element architecture, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object. You can also use the `Architecture` property of the `systemcomposer.arch.Component` object or the `ArchitecturePort` property of the `systemcomposer.arch.ComponentPort` object.

Example: `arch`

Example: `comp.Architecture`

Example: `conn`

Example: `compPort.ArchitecturePort`

## Output Arguments

### **propNames** — Property names

string array

Property names, returned as a string array, each in the form "`<profile>.<stereotype>.<property>`".

Data Types: `string`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”

Term	Definition	Application	More Information
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2019a

### See Also

`setProperty` | `getProperty` | `getEvaluatedPropertyValue` | `getPropertyValue`

### Topics

"Write Analysis Function"

# getStereotypes

**Package:** systemcomposer.arch

Get stereotypes applied on element of architecture model

## Syntax

```
stereotypes = getStereotypes(element)
```

## Description

`stereotypes = getStereotypes(element)` gets an array of fully qualified stereotype names that have been applied on an element of an architecture model.

## Examples

### Get Stereotypes

Create a model with a component.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
comp = addComponent(arch, "Component");
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

Apply the stereotype to the component and get the stereotypes on the component.

```
comp.applyStereotype("LatencyProfile.LatencyBase");
stereotypes = getStereotypes(comp)
```

```
stereotypes =
```

```
    1x1 cell array
```

```
    {'LatencyProfile.LatencyBase'}
```

## Input Arguments

### element — Architectural element

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

## Output Arguments

### stereotypes — List of stereotypes

cell array of character vectors

List of stereotypes, returned as a cell array of character vectors in the form '`<profile>.<stereotype>`'.

Data Types: `char`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”



Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• "Set Properties"</li> <li>• "Add Properties with Stereotypes"</li> <li>• "Set Properties for Analysis"</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• "Define Profiles and Stereotypes"</li> <li>• "Use Stereotypes and Profiles"</li> </ul>

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	“Define Physical Ports on Component”

Term	Definition	Application	More Information
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	“Architecture Model with Simscape Behavior for a DC Motor”
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on Ports”
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	“Describe Component Behavior Using Simscape”

## Version History

Introduced in R2019a

### See Also

`applyStereotype` | `removeStereotype` | `batchApplyStereotype` | `getStereotypeProperties`

### Topics

“Use Stereotypes and Profiles”

## getSubGroup

**Package:** `systemcomposer.view`

Get subgroup in element group of view

### Syntax

```
subGroup = getSubGroup(elementGroup, subGroupName)
```

### Description

`subGroup = getSubGroup(elementGroup, subGroupName)` gets a subgroup, `subGroup`, named `subGroupName` within the element group `elementGroup` of an architecture view.

### Examples

#### Create and Get Subgroup in View

Open the keyless entry system example and create a view `newView`.

```
scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("newView");
```

Open the Architecture Views Gallery to see the new view `newView`.

```
model.openViews
```

Create a subgroup `myGroup`.

```
group = view.Root.createSubGroup("myGroup");
```

Get the subgroup `myGroup`.

```
getGroup = view.Root.getSubGroup("myGroup")
```

```
getGroup =
  ElementGroup with properties:
      Name: 'myGroup'
      UUID: 'a0f647f5-8f2b-4169-a40d-e084f4dee414'
      Elements: []
      SubGroups: [0x0 systemcomposer.view.ElementGroup]
```

### Input Arguments

**elementGroup — Element group**

element group object

Element group for view, specified as a `systemcomposer.view.ElementGroup` object.

### **subGroupName — Name of subgroup**

character vector | string

Name of subgroup, specified as a character vector or string.

Example: "myGroup"

Data Types: char | string

## **Output Arguments**

### **subGroup — Subgroup**

element group object

Subgroup, returned as a `systemcomposer.view.ElementGroup` object.

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"

Term	Definition	Application	More Information
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

openViews | createView | getView | deleteView | systemcomposer.view.ElementGroup | systemcomposer.view.View | createSubGroup | deleteSubGroup | addElement | removeElement

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”



# getValue

**Package:** systemcomposer.analysis

Get value of property from element instance

## Syntax

```
[value,unit] = getValue(instance,property)
```

## Description

[value,unit] = getValue(instance,property) obtains the property property of the instance instance and assigns it to the specified value value.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The instance refers to the element instance on which the iteration is being performed.

---

## Examples

### Get Mass Property Value

Load the small unmanned aerial vehicle (UAV) model, create an architecture instance, and get the mass property value of a nested component.

```
scExampleSmallUAV
model = systemcomposer.loadModel("scExampleSmallUAVModel");
instance = instantiate(model.Architecture,"UAVComponent","NewInstance");
[massValue,unit] = getValue(instance.Components(1).Components(1),...
"UAVComponent.OnboardElement.Mass")
```

```
massValue = 1.7000
```

```
unit =
'kg'
```

## Input Arguments

### instance — Element instance

architecture instance | component instance | port instance | connector instance

Element instance, specified as a systemcomposer.analysis.ArchitectureInstance, systemcomposer.analysis.ComponentInstance, systemcomposer.analysis.PortInstance, or systemcomposer.analysis.ConnectorInstance object.

### property — Property

character vector | string

Property, specified in the form "<profile>.<stereotype>.<property>".

Data Types: char | string

## Output Arguments

### value — Property value

double (default) | single | int64 | int32 | int16 | int8 | uint64 | uint32 | uint8 | boolean | string | enumeration class name

Property value, returned as a data type that depends on how the property is defined in the profile.

### unit — Property unit

character vector

Property unit, returned as a character vector that describes the unit of the property as defined in the profile.

Example: 'kg'

Data Types: char

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>

Term	Definition	Application	More Information
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"><li>• “Define Profiles and Stereotypes”</li><li>• “Use Stereotypes and Profiles”</li></ul>

## **Version History**

**Introduced in R2019a**

### **See Also**

setValue | hasValue | systemcomposer.analysis.Instance

### **Topics**

“Write Analysis Function”

“Modeling System Architecture of Small UAV”

# getQualifiedName

**Package:** systemcomposer.arch

Get model element qualified name

## Syntax

```
getQualifiedName(element)
```

## Description

`getQualifiedName(element)` gets the qualified name of the architecture model element `element`.

## Examples

### Get Qualified Name of Component

Create a component, `newComponent`, then get its qualified name.

```
model = systemcomposer.createModel("newModel", true);
rootArch = get(model, "Architecture");
newComponent = addComponent(rootArch, "newComponent");
name = getQualifiedName(newComponent)
```

```
name =
    'newModel/newComponent'
```

## Input Arguments

### **element** — Architecture model element

element object

Architecture model element, specified as a `systemcomposer.arch.Element` object.

An element object translates to a `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	"Implement Component Behavior Using Simscape"
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"

Term	Definition	Application	More Information
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2019a

### See Also

Component | Variant Component | Lookup

### Topics

"Compose Architectures Visually"  
 "Decompose and Reuse Components"  
 "Implement Component Behavior Using Simscape"



# getView

**Package:** systemcomposer.arch

Find architecture view

## Syntax

```
view = getView(model,name)
```

## Description

`view = getView(model,name)` finds the view `view` in the architecture model `model` with view name `name`.

## Examples

### Create and Get View

Open the keyless entry system example and create a view, `newView`.

```
scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("newView");
```

Open the Architecture Views Gallery to see `newView`.

```
model.openViews
```

Find the view.

```
foundView = model.getView("newView")
```

```
foundView =
```

```
  View with properties:
```

```

        Name: 'newView'
        Root: [1x1 systemcomposer.view.ElementGroup]
        Model: [1x1 systemcomposer.arch.Model]
        UUID: 'c8e0a278-0ae0-4c8a-aca5-7ef730dbb1db'
        Select: []
        GroupBy: {}
        Color: '#0072bd'
        Description: ''
        IncludeReferenceModels: 1
```

## Input Arguments

**model** — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

**name — Name of view**

character vector | string

Name of view, specified as a character vector or string.

Example: "All Components Grouped by Review Status"

Data Types: char | string

## Output Arguments

**view — Architecture view**

view object

Architecture view found, returned as a `systemcomposer.view.View` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

systemcomposer.view.View | createView | deleteView | openViews | systemcomposer.view.ElementGroup

### Topics

“Create Architecture Views Interactively”  
“Create Architectural Views Programmatically”

# HasConnector

**Package:** systemcomposer.query

Create query to select architectural elements with connector based on specified subconstraint

## Syntax

```
query = HasConnector(subconstraint)
```

## Description

query = HasConnector(subconstraint) creates a query query that the find and createView functions use to select architectural elements with a connector that satisfies the given subconstraint subconstraint.

## Examples

### Find All Components with Connectors with Stereotype

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Specify the LatencyProfile.NodeLatency stereotype as a component stereotype. Set the default connector stereotype.

```
nodeLatency.AppliesTo = "Component";
nodeLatency.setDefaultElementStereotype("Connector", "LatencyProfile.ConnectorLatency");
```

Create a model, apply the profile to the model, and add a parent component. Apply the parent component stereotype on the parent component. Then, open the **Profile Editor**.

```

modelName = "archModel";
arch = systemcomposer.createModel(modelName,true);
arch.applyProfile("LatencyProfile");
newComponent = addComponent(arch.Architecture,"Component");
newComponent.applyStereotype("LatencyProfile.NodeLatency");
systemcomposer.profile.editor(profile)

```

Create two child components. Add ports. Then, create a connection between the ports and get stereotypes on the connector.

```

childComponent1 = addComponent(newComponent.Architecture,"Child1");
childComponent2 = addComponent(newComponent.Architecture,"Child2");

outPort1 = addPort(childComponent1.Architecture,"testSig","out");
inPort1 = addPort(childComponent2.Architecture,"testSig","in");
srcPort = getPort(childComponent1,"testSig");
destPort = getPort(childComponent2,"testSig");

connector = connect(srcPort,destPort);
stereotypes = getStereotypes(connector)

```

```
stereotypes =
```

```
  1×1 cell array
```

```
    {'LatencyProfile.ConnectorLatency'}
```

Create a query for all the elements with connectors with the ConnectorLatency stereotype and run the query.

```

constraint = HasConnector(HasStereotype(Property("Name") == "ConnectorLatency"));
baseComp = find(arch,constraint,Recurse=true,IncludeReferenceModels=true)

baseComp =

  1×1 cell array

    {'archModel/Component'}

```

## Input Arguments

### subconstraint — Condition restricting the query

query constraint object

Condition restricting the query, specified as a `systemcomposer.query.Constraint` object.

## Output Arguments

### query — Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”



Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2020a

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `HasInterface` | `HasPort` | `HasInterfaceElement` | `getQualifiedName`

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”

## HasInterface

**Package:** systemcomposer.query

Create query to select architectural elements with interface on port based on specified subconstraint

### Syntax

```
query = HasInterface(subconstraint)
```

### Description

`query = HasInterface(subconstraint)` creates a query `query` that the `find` and `createView` functions use to select architectural elements with an interface that satisfies the given subconstraint `subconstraint`.

### Examples

#### Construct Query to Select All Port Interfaces

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
sckeylessentrysystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query for all the interfaces in a port with `KeyFOBPosition` in the `Name` and run the query.

```
constraint = HasPort(HasInterface(contains(Property("Name"), "KeyFOBPosition")));
portInterfaces = find(model, constraint, Recurse=true, IncludeReferenceModels=true)
```

```
portInterfaces = 10x1 cell
    {'KeylessEntryArchitecture/Door Lock//Unlock System' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
    {'KeylessEntryArchitecture/Engine Control System' }
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller' }
    {'KeylessEntryArchitecture/FOB Locator System' }
    {'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module' }
    {'KeylessEntryArchitecture/Lighting System' }
    {'KeylessEntryArchitecture/Lighting System/Lighting Controller' }
    {'KeylessEntryArchitecture/Sound System' }
    {'KeylessEntryArchitecture/Sound System/Sound Controller' }
```

## Input Arguments

### subconstraint — Condition restricting the query

query constraint object

Condition restricting the query, specified as a `systemcomposer.query.Constraint` object.

## Output Arguments

### query — Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `HasPort` | `HasConnector` | `HasInterfaceElement` | `getQualifiedName`

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”

# HasInterfaceElement

**Package:** systemcomposer.query

Create query to select architectural elements with interface element on interface based on specified subconstraint

## Syntax

```
query = HasInterfaceElement(subconstraint)
```

## Description

query = HasInterfaceElement(subconstraint) creates a query query that the find and createView functions use to select architectural elements with an interface element that satisfies the given subconstraint subconstraint.

## Examples

### Construct Query to Select All Interface Elements

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
sckeylessEntrySystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query for all the interface elements with c in the Name and run the query.

```
constraint = HasPort(HasInterface(HasInterfaceElement(contains(Property("Name"), "c"))));
elements = find(model, constraint, Recurse=true, IncludeReferenceModels=true)
```

```
elements =
```

```
    0×0 empty cell array
```

## Input Arguments

**subconstraint** — Condition restricting the query

query constraint object

Condition restricting the query, specified as a systemcomposer.query.Constraint object.

## Output Arguments

### query — Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

createView | find | systemcomposer.query.Constraint | HasInterface | HasPort | HasConnector | getQualifiedName

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”

## HasPort

**Package:** systemcomposer.query

Create query to select architectural elements with port based on specified subconstraint

### Syntax

```
query = HasPort(subconstraint)
```

### Description

`query = HasPort(subconstraint)` creates a query `query` that the `find` and `createView` functions use to select architectural elements with a port that satisfies the given subconstraint `subconstraint`.

### Examples

#### Construct Query to Select All Sensor Component Ports

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
scKeylessEntrySystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query for all the elements with ports containing Sensor in the Name and run the query.

```
constraint = HasPort(contains(Property("Name"), "Sensor"));  
sensorComp = find(model, constraint, Recurse=true, IncludeReferenceModels=true)
```

```
sensorComp = 1x1 cell array  
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller'}
```

### Input Arguments

#### subconstraint — Condition restricting the query

query constraint object

Condition restricting the query, specified as a `systemcomposer.query.Constraint` object.



## Output Arguments

### query — Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `HasInterface` | `HasInterfaceElement` | `getQualifiedName` | `HasConnector`

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”

# hasProperty

**Package:** systemcomposer.arch

Find if element has property

## Syntax

```
result = hasProperty(element,property)
```

## Description

`result = hasProperty(element,property)` returns true if the property property has been added on the model element element.

## Examples

### Find Property on Component

Get the weight property from a component with the sysComponent stereotype applied.

Create a model with a component called Component.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
comp = addComponent(arch,"Component");
```

Create a profile with a stereotype and a property, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("sysProfile");
base = profile.addStereotype("sysComponent");
base.addProperty("weight",Type="double",DefaultValue="10",Units="g");
model.applyProfile("sysProfile")
```

Apply the stereotype to the component, then set a new weight property.

```
applyStereotype(comp,"sysProfile.sysComponent")
setProperty(comp,"sysProfile.sysComponent.weight","5","g")
```

Find if the weight property exists on the component.

```
result = hasProperty(comp,"sysProfile.sysComponent.weight")
```

```
result =
    logical
```

1

## Input Arguments

### element — Architectural element

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### property — Property

character vector | string

Property, specified as a character vector or string in the form "`<profile>.<stereotype>.<property>`".

Data Types: `char` | `string`

## Output Arguments

### result — Query result

true or 1 | false or 0

Query result, returned as a logical.

Data Types: `logical`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• "Set Properties"</li> <li>• "Add Properties with Stereotypes"</li> <li>• "Set Properties for Analysis"</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• "Define Profiles and Stereotypes"</li> <li>• "Use Stereotypes and Profiles"</li> </ul>

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	“Define Physical Ports on Component”



Term	Definition	Application	More Information
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2021a

### See Also

`addProperty` | `removeProperty` | `hasStereotype`

### Topics

"Use Stereotypes and Profiles"

## hasStereotype

**Package:** systemcomposer.arch

Find if element has stereotype applied

### Syntax

```
result = hasStereotype(element, stereotype)
```

### Description

`result = hasStereotype(element, stereotype)` returns true if the stereotype `stereotype` has been applied on the model element `element`.

### Examples

#### Apply Stereotype and Find Applied Stereotypes

Create a model with a component.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
comp = addComponent(arch, "Component");
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

Apply the stereotype to the component. Find if the stereotypes are applied on the component.

```
comp.applyStereotype("LatencyProfile.LatencyBase");
result = hasStereotype(comp, "LatencyProfile.LatencyBase")
```

```
result =
```

```
    logical
```

```
    1
```

### Input Arguments

**element** — Architectural element

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### stereotype — Stereotype

character vector | string | stereotype object

Stereotype, specified as a character vector or string in the form "<profile>.<stereotype>" or a `systemcomposer.profile.Stereotype` object.

Data Types: char | string

## Output Arguments

### result — Query result

true or 1 | false or 0

Query result, returned as a logical.

Data Types: logical

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• "Set Properties"</li> <li>• "Add Properties with Stereotypes"</li> <li>• "Set Properties for Analysis"</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• "Define Profiles and Stereotypes"</li> <li>• "Use Stereotypes and Profiles"</li> </ul>

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	“Define Physical Ports on Component”

Term	Definition	Application	More Information
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	“Architecture Model with Simscape Behavior for a DC Motor”
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on Ports”
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	“Describe Component Behavior Using Simscape”

## Version History

Introduced in R2021a

### See Also

`removeStereotype` | `applyStereotype` | `hasProperty` | `getStereotypes`

### Topics

“Use Stereotypes and Profiles”



# HasStereotype

**Package:** `systemcomposer.query`

Create query to select architectural elements with stereotype based on specified subconstraint

## Syntax

```
query = HasStereotype(subconstraint)
```

## Description

`query = HasStereotype(subconstraint)` creates a query `query` that the `find` and `createView` functions use to select architectural elements with a stereotype that satisfies the given subconstraint `subconstraint`.

## Examples

### Construct Query to Select All Hardware Components

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
sckeylessentrysystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query for all the hardware components and run the query, displaying one of them.

```
constraint = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent"));
hwComp = find(model, constraint, Recurse=true, IncludeReferenceModels=true);
comp = hwComp(16)
```

```
comp = 1x1 cell array
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Sensor'}
```

## Input Arguments

**subconstraint** — Condition restricting the query

query constraint object

Condition restricting the query, specified as a `systemcomposer.query.Constraint` object.

## Output Arguments

### query — Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `IsStereotypeDerivedFrom` | `getQualifiedName`

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”

## hasValue

**Package:** systemcomposer.analysis

Find if element instance has property value

### Syntax

```
result = hasValue(instance,property)
```

### Description

`result = hasValue(instance,property)` queries whether the instance `instance` has the given property `property`.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

### Examples

#### Query Whether Instance Has Property

Load the small unmanned aerial vehicle (UAV) model, create an architecture instance, and query whether an instance element has a property included.

```
scExampleSmallUAV
model = systemcomposer.loadModel("scExampleSmallUAVModel");
instance = instantiate(model.Architecture,"UAVComponent","NewInstance");
queryResult = hasValue(instance.Components(1).Components(1),...
"UAVComponent.OnboardElement.Mass")

queryResult = logical
    1
```

### Input Arguments

#### **instance** — Element instance

architecture instance | component instance | port instance | connector instance

Element instance, specified as a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

#### **property** — Property

character vector | string

Property, specified in the form "<profile>.<stereotype>.<property>".

Data Types: char | string

## Output Arguments

### result – Query result

true or 1 | false or 0

Query result, returned as a logical.

Data Types: logical

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”

Term	Definition	Application	More Information
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

**See Also**

setValue | getValue | systemcomposer.analysis.Instance

**Topics**

“Write Analysis Function”

“Modeling System Architecture of Small UAV”

## systemcomposer.importModel

Import model information from MATLAB tables

### Syntax

```
archModel = systemcomposer.importModel(modelName,components,ports,
connections,portInterfaces,requirementLinks,parameters)
archModel = systemcomposer.importModel(modelName,importStruct)
[archModel,idMappingTable,importLog,errorLog] = systemcomposer.importModel(
___)
```

### Description

`archModel = systemcomposer.importModel(modelName,components,ports,connections,portInterfaces,requirementLinks,parameters)` creates a new architecture model based on MATLAB tables that specify components, ports, connections, port interfaces, requirement links, and parameters. The only required input arguments are `modelName` and the `components` table. For empty table input arguments, enter `table.empty`. However, trailing empty tables are ignored and do not need to be entered. To import a basic architecture model, see “Define Basic Architecture”. To import `requirementLinks`, you need a Requirements Toolbox license.

`archModel = systemcomposer.importModel(modelName,importStruct)` creates a new architecture model based on a structure of MATLAB tables that have prescribed formats to specify model element relationships, stereotypes, and properties. For more information on the import structure, see “Import and Export Architecture Models”.

`[archModel,idMappingTable,importLog,errorLog] = systemcomposer.importModel( ___)` creates a new architecture model with output arguments `idMappingTable` with table information, `importLog` to display import information, and `errorLog` to display import error information. All previous syntax descriptions are included.

### Examples

#### Import and Export Architectures

In System Composer™, an architecture is fully defined by three sets of information:

- Component information
- Port information
- Connection information

You can import an architecture into System Composer when this information is defined in or converted into MATLAB® tables.

In this example, the architecture information of a simple unmanned aerial vehicle (UAV) system is defined in a Microsoft® Excel® spreadsheet and is used to create a System Composer architecture model. It also links elements to the specified system level requirement. You can modify the files in this example to import architectures defined in external tools, when the data includes the required



information. The example also shows how to export this architecture information from System Composer architecture model to an Excel spreadsheet.

### Architecture Definition Data

You can characterize the architecture as a network of components and import by defining components, ports, connections, interfaces and requirement links in MATLAB tables. The `components` table must include name, unique ID, and parent component ID for each component. The spreadsheet can also include other relevant information required to construct the architecture hierarchy for referenced model, and stereotype qualifier names. The `ports` table must include port name, direction, component, and port ID information. Port interface information may also be required to assign ports to components. The `connections` table includes information to connect ports. At a minimum, this table must include the connection ID, source port ID, and destination port ID.

The `systemcomposer.importModel(importModelName)` function:

- Reads stereotype names from the `components` table and loads the profiles
- Creates components and attaches ports
- Creates connections using the connection map
- Sets interfaces on ports
- Links elements to specified requirements (requires a Requirements Toolbox™ license)
- Saves referenced models
- Saves the architecture model

Instantiate adapter class to read from Excel.

```
modelName = "simpleUAVArchitecture";
```

`ImportModelFromExcel` function reads the Excel file and creates the MATLAB tables.

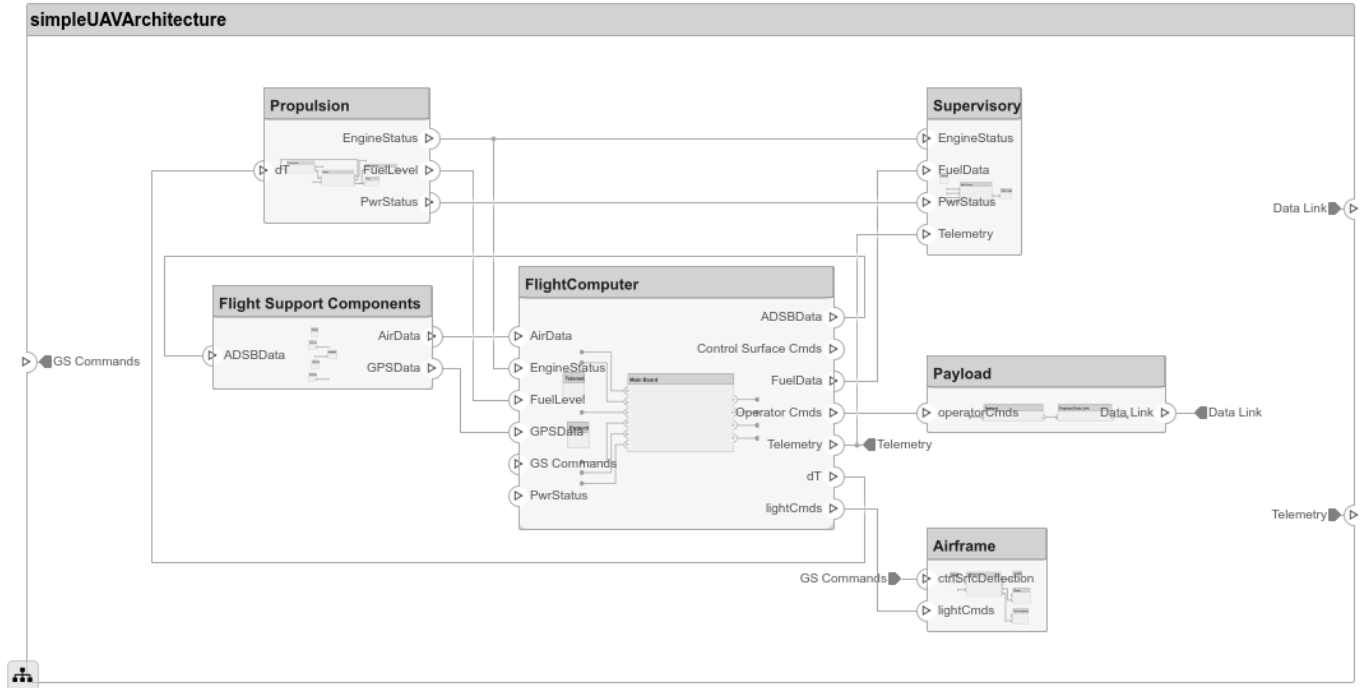
```
importAdapter = ImportModelFromExcel("SmallUAVModel.xls", "Components", ...
    "Ports", "Connections", "PortInterfaces", "RequirementLinks");
importAdapter.readTableFromExcel();
```

### Import an Architecture

```
model = systemcomposer.importModel(modelName, importAdapter.Components, ...
    importAdapter.Ports, importAdapter.Connections, importAdapter.Interfaces, ...
    importAdapter.RequirementLinks);
```

Auto-arrange blocks in the generated model.

```
Simulink.BlockDiagram.arrangeSystem(modelName)
```



## Export an Architecture

You can export an architecture to MATLAB tables and then convert the tables to an external file.

```
exportedSet = systemcomposer.exportModel(modelName);
```

The output of the function is a structure that contains the component table, port table, connection table, the interface table, and the requirement links table. Save this structure to an Excel file.

```
SaveToExcel("ExportedUAVModel",exportedSet);
```

## Input Arguments

### modelName — Name of model

character vector | string

Name of model, specified as a character vector or string.

Example: "exMobileRobot"

Data Types: char | string

### components — Model component information

MATLAB table

Model component information, specified as a MATLAB table. The component table must include the columns Name, ID, and ParentID. To specify ComponentType as Variant, Composition (default), StateflowBehavior, or Behavior (reference components and subsystem components) and to set a ReferenceModelName, see "Import Variant Components, Stateflow Behaviors, or Reference Components". To apply stereotypes using StereotypeNames and set property values to components, see "Apply Stereotypes and Set Property Values on Imported Model".

Data Types: table

### **ports — Model port information**

MATLAB table

Model port information, specified as a MATLAB table. The ports table must include the columns `Name`, `Direction`, `ID`, and `CompID`. The `Direction` column can have values `Input`, `Output`, or `Physical`. The optional column `InterfaceID` specifies the interface. `portInterfaces` information may also be required to assign interfaces to ports.

Data Types: table

### **connections — Model connections information**

MATLAB table

Model connections information, specified as a MATLAB table. The connections table must include the columns `Name`, `ID`, `SourcePortID`, and `DestPortID`. To specify `SourceElement` or `DestinationElement` on an architecture port, see “Specify Elements on Architecture Port”. Assign a stereotype using the optional column `StereotypeNames`. The optional `Kind` column can be specified as the default `Data` or `Physical` for physical connections.

Data Types: table

### **portInterfaces — Model port interfaces information**

MATLAB table

Model port interfaces information, specified as a MATLAB table. The port interfaces table must include the columns `Name`, `ID`, `ParentID`, `DataType`, `Dimensions`, `Units`, `Complexity`, `Minimum`, and `Maximum`. To import interfaces and map ports to interfaces, see “Import Data Interfaces and Map Ports to Interfaces”. Add a description using the option column `Description`. Assign a stereotype using the optional column `StereotypeNames`.

Data Types: table

### **requirementLinks — Model requirement links information**

MATLAB table

Model requirement links information, specified as a MATLAB table. The requirement links table must include the columns `Label`, `ID`, `SourceID`, `DestinationType`, `DestinationID`, and `Type`. For an example, see “Assign Requirement Links on Imported Model”. To update reference requirement links from an imported file and integrate them into the model, see “Update Reference Requirement Links from Imported File” on page 4-791. Optional columns include: `DestinationArtifact`, `SourceArtifact`, `ReferencedReqID`, `Keywords`, `CreatedOn`, `CreatedBy`, `ModifiedOn`, `ModifiedBy`, and `Revision`. A Requirements Toolbox license is required to import the `requirementLinks` table to a System Composer architecture model.

Data Types: table

### **parameters — Model parameters information**

MATLAB table

Model parameters information, specified as a MATLAB table. The parameters table must include the columns `Name`, `ID`, `Parent`, and `Value`. To import an architecture model with parameters programmatically, see Import Architecture With Parameters. Add value type information to the parameter with the `Units`, `Type`, `Complexity`, `Minimum`, and `Maximum` columns. Promote

parameters to an architecture in the hierarchy using the `PromotedTo` column. For more information, see “Import Parameters with Parameter Value Types”.

Data Types: `table`

### **importStruct – Model tables**

structure

Model tables, specified as a structure containing the `tables` components, `ports`, `connections`, `portInterfaces`, `requirementLinks`, and `parameters` and a field `domain`. Only the `components` table is required. Possible values for `domain` are the default “System” for architecture models and “Software” for software architecture models. For software architecture models, to import a model with functions, the `importStruct` can have a `functions` field that contains function information.

For more information on the import structure, see “Import and Export Architecture Models”.

Data Types: `struct`

## **Output Arguments**

### **archModel – Handle to architecture model**

architecture object

Handle to architecture model, specified as a `systemcomposer.arch.Architecture` object.

### **idMappingTable – Mapping of custom IDs and internal UUIDs of elements**

structure

Mapping of custom IDs and internal UUIDs of elements, returned as a `struct` of MATLAB tables.

Data Types: `struct`

### **importLog – Confirmation that elements were imported**

cell array of character vectors

Confirmation that elements were imported, returned as a cell array of character vectors.

Data Types: `char`

### **errorLog – Errors reported during import process**

cell array of message objects

Errors reported during import process, returned as a cell array of message objects. You can obtain the error text by calling the `getString` method on each message object. For example, `errorLog.getString` is used to obtain the errors reported as a string.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

#### Blocks

Component | Variant Component | Reference Component

#### Functions

exportModel | systemcomposer.exportToVersion |  
systemcomposer.updateLinksToReferenceRequirements

#### Topics

"Import and Export Architecture Models"  
"Author Parameters in System Composer Using Parameter Editor"

# increaseExecutionOrder

**Package:** systemcomposer.arch

Change function execution order to later

## Syntax

```
increaseExecutionOrder(functionObj)
```

## Description

`increaseExecutionOrder(functionObj)` increases execution order of the specified function `functionObj` by 1. If the function is at the maximum execution order, the `increaseExecutionOrder` method will fail with a warning.

## Examples

### Change Execution Order of Software Functions

This example shows the software architecture of a throttle position control system and how to schedule the execution order of the root level functions.

```
model = systemcomposer.openModel("ThrottleControlComposition");
```

Simulate the model to populate it with functions.

```
sim("ThrottleControlComposition");
```

View the function names ordered by execution order.

```
functions = {model.Architecture.Functions.Name}'
```

```
functions = 6x1 cell
    {'Actuator_output_5ms' }
    {'Controller_run_5ms' }
    {'TPS_Primary_read_5ms' }
    {'TPS_Secondary_read_5ms' }
    {'TP_Monitor_D1' }
    {'APP_Sensor_read_10ms' }
```

Decrease the execution order of the third function.

```
decreaseExecutionOrder(model.Architecture.Functions(3))
```

View the function names ordered by execution order.

```
functions = {model.Architecture.Functions.Name}'
```

```
functions = 6x1 cell
    {'Actuator_output_5ms' }
    {'TPS_Primary_read_5ms' }
```

```

{'Controller_run_5ms'   }
{'TPS_Secondary_read_5ms'}
{'TP_Monitor_D1'      }
{'APP_Sensor_read_10ms'}

```

The third function is now moved up in execution order, executing earlier.

Increase the execution order of the second function.

```
increaseExecutionOrder(model.Architecture.Functions(2))
```

View the function names ordered by execution order.

```

functions = {model.Architecture.Functions.Name}'

functions = 6x1 cell
    {'Actuator_output_5ms'   }
    {'Controller_run_5ms'   }
    {'TPS_Primary_read_5ms' }
    {'TPS_Secondary_read_5ms'}
    {'TP_Monitor_D1'      }
    {'APP_Sensor_read_10ms'}

```

The second function is now moved down in execution order, executing later.

## Input Arguments

### functionObj – Function

function object

Function, specified as a `systemcomposer.arch.Function` object.

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>



Term	Definition	Application	More Information
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>• “Author Service Interfaces for Client-Server Communication”</li> <li>• <code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

## **Version History**

**Introduced in R2021b**

### **See Also**

`systemcomposer.createModel` | `createArchitectureModel` | `decreaseExecutionOrder`

### **Topics**

“Modeling Software Architecture of Throttle Position Control System”

“Simulate and Deploy Software Architectures”

“Author Software Architectures”

## inlineComponent

**Package:** systemcomposer.arch

Remove reference architecture or behavior from component

### Syntax

```
componentObj = inlineComponent(component,inlineFlag)
```

### Description

`componentObj = inlineComponent(component,inlineFlag)` retains the contents of the architecture model referenced by the specified `component` and breaks the link to the reference model. If `inlineFlag` is set to 0 (`false`), then the contents of the architecture model are removed and only interfaces remain. You can also use `inlineComponent` to remove Stateflow chart and Simulink behaviors from a component or to remove Simulink model or subsystem behaviors referenced by a component.

### Examples

#### Reuse Component and Remove Architecture Reference

Save the component `robotComp` in the architecture model `Robot.slx` and reference it from another component, `electricComp`, so that the `electricComp` component uses the architecture of the `robotComp` component. Remove the architecture reference from the `robotComp` component so that its architecture can be edited independently.

Create a model `archModel.slx`.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
```

Add two components to the model with the names `"electricComp"` and `"robotComp"`.

```
names = ["electricComp","robotComp"];
comp = addComponent(arch,names);
```

Save the `robotComp` component in the `Robot.slx` model so the component references the model.

```
saveAsModel(comp(2),"Robot");
```

Link the `electricComp` component to the same model `Robot.slx` so it uses the architecture of the original `robotComp` component and references the architecture model `Robot.slx`.

```
linkToModel(comp(1),"Robot");
```

Remove the architecture reference from the `robotComp` component while retaining the contents, so that its architecture can be edited independently, breaking the link to the referenced model.

```
inlineComponent(comp(2),true);
```

Clean up the model.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

### Add Stateflow Behavior to Component and Remove

Add a Stateflow chart behavior to the component named `robotComp` within the current model. Then, remove the behavior.

Create a model `archModel.slx`.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
```

Add two components to the model with the names `"electricComp"` and `"robotComp"`.

```
names = ["electricComp", "robotComp"];
comp = addComponent(arch, names);
```

Add Stateflow chart behavior model to the `robotComp` component.

```
createStateflowChartBehavior(comp(2));
```

Remove Stateflow chart behavior from the `robotComp` component and remove all contents of the Stateflow chart.

```
inlineComponent(comp(2), false);
```

Clean up the model.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

## Input Arguments

### **component** — Component

component object

Component linked to an architecture model, specified as a `systemcomposer.arch.Component` object.

### **inlineFlag** — Control of contents of component

true or 1 | false or 0

Control of contents of component, specified as a logical 1 (`true`) if contents of the referenced architecture model are copied to the component architecture and 0 (`false`) if the contents are not copied and only ports and interfaces are preserved.

Data Types: `logical`

## Output Arguments

### **componentObj** — Component

component object

Component with referenced architecture or behavior removed, returned as a `systemcomposer.arch.Component` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• "Implement Component Behavior Using Simulink"</li> <li>• "Create Architecture Reference"</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• "Author Parameters in System Composer Using Parameter Editor"</li> <li>• "Access Model Arguments as Parameters on Reference Components"</li> <li>• "Use Parameters to Store Instance Values with Components"</li> </ul>

Term	Definition	Application	More Information
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2019a

### See Also

`createSimulinkBehavior` | `createArchitectureModel` | `createStateflowChartBehavior` | `extractArchitectureFromSimulink` | `isReference` | Reference Component

### Topics

“Implement Component Behavior Using Simulink”

“Decompose and Reuse Components”

“Implement Component Behavior Using Stateflow Charts”

“Create Simulink Subsystem Behavior Using Subsystem Component”

“Simulate and Deploy Software Architectures”



# instantiate

**Package:** systemcomposer.arch

Create analysis instance from specification

## Syntax

```
instance = instantiate(arch,properties,name)
instance = instantiate(arch,profile,name)
instance = instantiate( ____,Name,Value)
```

## Description

`instance = instantiate(arch,properties,name)` creates an instance `instance` named `name` of a model architecture `arch` with properties `properties` for analysis. Get the `Architecture` property of the `systemcomposer.arch.Model` object `model` using `model.Architecture` in the MATLAB Command Window.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

`instance = instantiate(arch,profile,name)` creates an instance `instance` named `name` of a model architecture `arch` with all stereotypes in a profile `profile` for analysis.

`instance = instantiate( ____,Name,Value)` creates an instance of a model architecture for analysis with additional arguments.

## Examples

### Instantiate All Properties of Stereotypes in Profile

Instantiate all properties of stereotypes in a profile that will be applied to specific elements during instantiation.

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");
```

```
portLatency = profile.addStereotype("PortLatency", ...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth", Type="double");
portLatency.addProperty("dummy", Type="int32");

profile.save
```

Create a new model and apply the profile to the model.

```
model = systemcomposer.createModel("archModel", true);
model.applyProfile("LatencyProfile");
```

Specify type of elements each stereotype can be applied on.

```
NodeLatency = struct("elementKinds", ["Component"]);
ConnectorLatency = struct("elementKinds", ["Connector"]);
LatencyBase = struct("elementKinds", ["Connector", "Port", "Component"]);
PortLatency = struct("elementKinds", ["Port"]);
```

Create the analysis structure.

```
LatencyAnalysis = struct("NodeLatency", NodeLatency, ...
    "ConnectorLatency", ConnectorLatency, ...
    "PortLatency", PortLatency, ...
    "LatencyBase", LatencyBase);
```

Create the properties structure.

```
properties = struct("LatencyProfile", LatencyAnalysis);
```

Instantiate all properties of stereotypes in the profile.

```
instance = instantiate(model.Architecture, properties, "NewInstance")
```

## Instantiate Specific Properties of Stereotypes in Profile

Instantiate specific properties of stereotypes in a profile that will be applied to specific elements during instantiation.

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency", ...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure", Type="boolean");
connLatency.addProperty("linkDistance", Type="double");

nodeLatency = profile.addStereotype("NodeLatency", ...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");

portLatency = profile.addStereotype("PortLatency", ...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth", Type="double");
portLatency.addProperty("dummy", Type="int32");

profile.save
```

Create a new model and apply the profile to the model.

```
model = systemcomposer.createModel("archModel",true);
model.applyProfile("LatencyProfile");
```

Specify some properties of the stereotypes in the profile.

```
NodeLatency = struct("elementKinds",["Component"], ...
    "properties",struct("resources",true));
ConnectorLatency = struct("elementKinds",["Connector"], ...
    "properties",struct("secure",true,"linkDistance",true));
LatencyBase = struct("elementKinds",[], ...
    "properties",struct("dataRate",true,"latency",false));
PortLatency = struct('elementKinds',["Port"], ...
    "properties",struct("queueDepth",true));

LatencyAnalysis = struct("NodeLatency",NodeLatency, ...
    "ConnectorLatency",ConnectorLatency, ...
    "PortLatency",PortLatency, ...
    "LatencyBase",LatencyBase);
```

Create the properties structure.

```
properties = struct("LatencyProfile",LatencyAnalysis);
```

Instantiate some properties of stereotypes in the profile.

```
instance = instantiate(model.Architecture,properties,"NewInstance")
```

## Instantiate All Stereotypes in Profile

Instantiate all stereotypes already in a profile that will be applied to elements during instantiation.

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Create a new model and apply the profile to the model.

```
model = systemcomposer.createModel("archModel",true);
model.applyProfile("LatencyProfile");
```

Instantiate all stereotypes in a profile.

```
instance = instantiate(model.Architecture, "LatencyProfile", "NewInstance")
```

### Analyze Latency Characteristics

Create an instantiation for analysis for a system with latency in its wiring. The materials used are copper, fiber, and WiFi.

### Create Latency Profile with Stereotypes and Properties

Create a System Composer profile with a base, connector, component, and port stereotype. Add properties with default values to each stereotype as needed for analysis.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfileC");
```

Add a base stereotype with properties.

```
latencybase = profile.addStereotype("LatencyBase");  
latencybase.addProperty("latency", Type="double");  
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");
```

Add a connector stereotype with properties.

```
connLatency = profile.addStereotype("ConnectorLatency", ...  
    Parent="LatencyProfileC.LatencyBase");  
connLatency.addProperty("secure", Type="boolean", DefaultValue="true");  
connLatency.addProperty("linkDistance", Type="double");
```

Add a component stereotype with properties.

```
nodeLatency = profile.addStereotype("NodeLatency", ...  
    Parent="LatencyProfileC.LatencyBase");  
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");
```

Add a port stereotype with properties.

```
portLatency = profile.addStereotype("PortLatency", ...  
    Parent="LatencyProfileC.LatencyBase");  
portLatency.addProperty("queueDepth", Type="double", DefaultValue="4.29");  
portLatency.addProperty("dummy", Type="int32");
```

### Instantiate Using Analysis Function

Create a new model and apply the profile. Create components, ports, and connections in the model. Apply stereotypes to the model elements. Finally, instantiate using the analysis function.

```
model = systemcomposer.createModel("archModel", true);  
arch = model.Architecture;
```

Apply profile to model.

```
model.applyProfile("LatencyProfileC");
```

Create components, ports, and connections.

```
componentSensor = addComponent(arch, "Sensor");  
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorPower'}, {'in', 'out'});
```

```

componentPlanning = addComponent(arch, "Planning");
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorPower', 'MotionCommand'});
componentMotion = addComponent(arch, "Motion");
motionPorts = addPort(componentMotion.Architecture, {'MotionCommand', 'MotionData'}, {'in', 'out'});

c_sensorData = connect(arch, componentSensor, componentPlanning);
c_motionData = connect(arch, componentMotion, componentSensor);
c_motionCommand = connect(arch, componentPlanning, componentMotion);

```

Clean up the canvas.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

Batch apply stereotypes to model elements.

```

batchApplyStereotype(arch, "Component", "LatencyProfileC.NodeLatency");
batchApplyStereotype(arch, "Port", "LatencyProfileC.PortLatency");
batchApplyStereotype(arch, "Connector", "LatencyProfileC.ConnectorLatency");

```

Instantiate using the analysis function.

```

instance = instantiate(model.Architecture, "LatencyProfileC", "NewInstance", ...
    Function=@calculateLatency, Arguments="3", ...
    Strict=true, NormalizeUnits=false, Direction="PreOrder")

```

instance =

ArchitectureInstance with properties:

```

    Specification: [1x1 systemcomposer.arch.Architecture]
        IsStrict: 1
    NormalizeUnits: 0
    AnalysisFunction: @calculateLatency
    AnalysisDirection: PreOrder
    AnalysisArguments: '3'
    ImmediateUpdate: 0
        Components: [1x3 systemcomposer.analysis.ComponentInstance]
            Ports: [0x0 systemcomposer.analysis.PortInstance]
        Connectors: [1x3 systemcomposer.analysis.ConnectorInstance]
            Name: 'NewInstance'

```

## Inspect Component, Port, and Connector Instances

Get properties from component, port, and connector instances.

```
defaultResources = instance.Components(1).getValue("LatencyProfileC.NodeLatency.resources")
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue("LatencyProfileC.ConnectorLatency.secure")
```

```
defaultSecure = logical
```

```
1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue("LatencyProfileC.PortLatency.queueDepth")
```

```
defaultQueueDepth = 4.2900
```

## Input Arguments

### **arch** — Architecture

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **properties** — Stereotype properties

structure

Stereotype properties, specified as a structure containing profile, stereotype, and property information. Use `properties` to specify which stereotypes and properties need to be instantiated.

Data Types: `struct`

### **name** — Name of instance

character vector | string

Name of instance generated from the model, specified as a character vector or string.

Example: "NewInstance"

Data Types: `char` | `string`

### **profile** — Profile name

character vector | string

Profile name, specified as a character vector or string.

Example: 'LatencyProfile'

Data Types: `char` | `string`

## Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example:

```
instantiate(model.Architecture,"LatencyProfile","NewInstance",Function=@calculateLatency,Arguments="3",Strict=true,NormalizeUnits=false,Direction="PreOrder")
```

### **NormalizeUnits** — Whether to normalize value based on units

false or 0 (default) | true or 1

Whether to normalize value based on units, if any, specified in property definition upon instantiation, specified as a logical.

Example:

```
instantiate(model.Architecture,'LatencyProfile','NewInstance','NormalizeUnits',true)
```

Data Types: `logical`

### **Function — Analysis function**

function handle

Analysis function, specified as the MATLAB function handle to be executed when analysis is run.

### **Arguments — Analysis arguments**

cell array of character vectors | array of strings | character vector | string

Analysis arguments, specified as a character vector, string, array of strings, or a cell array of character vectors of optional arguments to the analysis function.

Data Types: `char` | `string`

### **Direction — Iteration type**

"PreOrder" | "PostOrder" | "TopDown" | "BottomUp"

Iteration type, specified as "PreOrder", "PostOrder", "TopDown", or "BottomUp".

- **Pre-order** — Start from the top level, move to a child component, and process the subcomponents of that component recursively before moving to a sibling component.
- **Top-Down** — Like pre-order, but process all sibling components before moving to their subcomponents.
- **Post-order** — Start from components with no subcomponents, process each sibling, and then move to parent.
- **Bottom-up** — Like post-order, but process all subcomponents at the same depth before moving to their parents.

Data Types: `char` | `string`

### **Strict — Condition for instances getting properties**

false or 0 (default) | true or 1

Condition for instances getting properties only if the specification of the instance has the stereotype applied, specified as a logical.

Data Types: `logical`

## **Output Arguments**

### **instance — Architecture instance**

architecture instance object

Architecture instance, returned as a `systemcomposer.analysis.ArchitectureInstance` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”



Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"

Term	Definition	Application	More Information
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>• “Analyze Architecture Model with Analysis Function”</li> <li>• “Analyze Architecture”</li> <li>• “Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>• “Analysis Function Constructs”</li> <li>• “Write Analysis Function”</li> </ul>

Term	Definition	Application	More Information
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	"Run Analysis Function"
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	"Create a Model Instance for Analysis"

## Version History

Introduced in R2019a

### See Also

systemcomposer.analysis.Instance | deleteInstance | loadInstance | save | update | iterate

### Topics

"Write Analysis Function"

# isArchitecture

**Package:** systemcomposer.analysis

Find if instance is architecture instance

## Syntax

```
flag = isArchitecture(instance)
```

## Description

`flag = isArchitecture(instance)` finds whether the instance specified as `instance` is an architecture instance.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

## Examples

### Query Architecture Instance

Load the small unmanned aerial vehicle (UAV) model, create an architecture instance, and query whether the instance is an architecture instance.

```
scExampleSmallUAV
model = systemcomposer.loadModel("scExampleSmallUAVModel");
instance = instantiate(model.Architecture, "UAVComponent", "NewInstance");
flag = isArchitecture(instance)
```

```
flag = logical
      1
```

## Input Arguments

### **instance** — Element instance

architecture instance | component instance | port instance | connector instance

Element instance, specified as a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

## Output Arguments

### flag — Whether instance is architecture instance

true or 1 | false or 0

Whether instance is architecture instance `systemcomposer.analysis.ArchitectureInstance`, returned as a logical.

Data Types: `logical`

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

[systemcomposer.analysis.Instance](#) | [isComponent](#) | [isConnector](#) | [isPort](#)

### Topics

"Write Analysis Function"

"Modeling System Architecture of Small UAV"

# isComponent

**Package:** systemcomposer.analysis

Find if instance is component instance

## Syntax

```
flag = isComponent(instance)
```

## Description

`flag = isComponent(instance)` finds whether the instance specified by `instance` is a component instance.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

## Examples

### Query Component Instance

Load the small unmanned aerial vehicle (UAV) model, create an architecture instance, and query whether the instance modified by the `Components` property is a component instance.

```
scExampleSmallUAV
model = systemcomposer.loadModel("scExampleSmallUAVModel");
instance = instantiate(model.Architecture, "UAVComponent", "NewInstance");
flag = isComponent(instance.Components(1))
```

```
flag = logical
      1
```

## Input Arguments

### instance — Element instance

architecture instance | component instance | port instance | connector instance

Element instance, specified as a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.



## Output Arguments

### flag — Whether instance is component instance

true or 1 | false or 0

Whether instance is component instance `systemcomposer.analysis.ComponentInstance`, returned as a logical.

Data Types: `logical`

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

isArchitecture | isConnector | isPort | systemcomposer.analysis.Instance

### Topics

"Write Analysis Function"

"Modeling System Architecture of Small UAV"

## isConnector

**Package:** systemcomposer.analysis

Find if instance is connector instance

### Syntax

```
flag = isConnector(instance)
```

### Description

`flag = isConnector(instance)` finds whether the instance specified by `instance` is a connector instance.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

## Examples

### Query Connector Instance

Load the small unmanned aerial vehicle (UAV) model, create an architecture instance, and query whether the instance modified by the `Connectors` property is a connector instance.

```
scExampleSmallUAV
model = systemcomposer.loadModel("scExampleSmallUAVModel");
instance = instantiate(model.Architecture, "UAVComponent", "NewInstance");
flag = isConnector(instance.Connectors(1))
```

```
flag = logical
      1
```

## Input Arguments

### instance — Element instance

architecture instance | component instance | port instance | connector instance

Element instance, specified as a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

## Output Arguments

### flag — Whether instance is connector instance

true or 1 | false or 0

Whether instance is connector instance `systemcomposer.analysis.ConnectorInstance`, returned as a logical.

Data Types: `logical`

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

`systemcomposer.analysis.Instance` | `isArchitecture` | `isComponent` | `isPort`

### Topics

"Write Analysis Function"

"Modeling System Architecture of Small UAV"

## IsInRange

**Package:** `systemcomposer.query`

Create query to select range of property values

### Syntax

```
query = IsInRange(name,beginRangeValue,endRangeValue)
```

### Description

`query = IsInRange(name,beginRangeValue,endRangeValue)` creates a query `query` that the `find` and `createView` functions use to select a range of values from `beginRangeValue` to `endRangeValue` for a specified property name `name`.

### Examples

#### Find Model Elements that Satisfy Property Range

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
sckeylessEntrySystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query to find components with values from 10 ms to 40 ms for the Latency property.

```
constraint = IsInRange(PropertyValue("AutoProfile.BaseComponent.Latency"),...
Value(10,"ms"),Value(40,"ms"));
```

```
latency = find(model,constraint,Recurse=true,IncludeReferenceModels=true)
```

```
latency = 5x1 cell
```

```
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Actuator' }
    {'KeylessEntryArchitecture/Sound System/Dashboard Speaker' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Actuator' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Actuator' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Actuator' }
```

### Input Arguments

**name — Property name**

character vector | string



Property name for model element, specified in the form "<profile>.<stereotype>.<property>" or any property on the designated class.

Example: "Name"

Example: "AutoProfile.BaseComponent.Latency"

Data Types: char

### **beginRangeValue – Beginning range value**

value object

Beginning range value for propertyName, specified as a systemcomposer.query.Value object.

Example: Value(20)

Example: Value(5, "ms")

### **endRangeValue – Ending range value**

value object

Ending range value for propertyName, specified as a systemcomposer.query.Value object.

Example: Value(100)

Example: Value(20, "ms")

## **Output Arguments**

### **query – Query**

query constraint object

Query, returned as a systemcomposer.query.Constraint object.

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	"Modeling System Architecture of Keyless Entry System"

Term	Definition	Application	More Information
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>“Create Architecture Views Interactively”</li> <li>“Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li><i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li><i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `getQualifiedName`

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”

# isPort

**Package:** systemcomposer.analysis

Find if instance is port instance

## Syntax

```
flag = isPort(instance)
```

## Description

`flag = isPort(instance)` finds whether the instance specified by `instance` is a port instance.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

## Examples

### Query Port Instance

Load the small unmanned aerial vehicle (UAV) model, create an architecture instance, and query whether the instance modified by the `Ports` property is a port instance.

```
scExampleSmallUAV
model = systemcomposer.loadModel("scExampleSmallUAVModel");
instance = instantiate(model.Architecture,"UAVComponent","NewInstance");
flag = isPort(instance.Ports(1))

flag = logical
      1
```

## Input Arguments

### instance — Element instance

architecture instance | component instance | port instance | connector instance

Element instance, specified as a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

### flag — Whether instance is port instance

true or 1 | false or 0

Whether instance is port instance `systemcomposer.analysis.PortInstance`, returned as a logical.

Data Types: logical

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>• “Analyze Architecture Model with Analysis Function”</li> <li>• “Analyze Architecture”</li> <li>• “Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>• “Analysis Function Constructs”</li> <li>• “Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## Version History

Introduced in R2019a

### See Also

isArchitecture | isComponent | isConnector | systemcomposer.analysis.Instance

### Topics

“Write Analysis Function”

“Modeling System Architecture of Small UAV”

# isProtected

**Package:** systemcomposer.arch

Find if component reference model is protected

## Syntax

```
flag = isProtected(compObj)
```

## Description

`flag = isProtected(compObj)` returns whether or not the referenced model on the component is protected. A protected model is saved with an SLXP extension.

## Examples

### Find If Component Reference Model is Protected

Find whether or not the referenced model on the component is protected.

Create a new System Composer model and add a new component.

```
model = systemcomposer.createModel("archModel");
rootArch = get(model,"Architecture");
newComponent = addComponent(rootArch,"newComponent");
```

Create new Simulink reference model and save.

```
newRef = new_system("newReference","Model");
save_system(newRef);
```

Protect the Simulink model reference.

```
Simulink.ModelReference.protect(newRef);
```

Link the Simulink model to the component newComponent.

```
linkToModel(newComponent,"newReference.slxp");
```

Verify that the reference model linked to the component is protected.

```
flag = isProtected(newComponent)
```

```
flag =
    logical
     1
```

## Input Arguments

### compObj — Component

component object | variant component object

Component, specified as a `systemcomposer.arch.Component` or `systemcomposer.arch.VariantComponent` object.

## Output Arguments

### **flag** – Whether referenced model on component is protected

true or 1 | false or 0

Whether referenced model on component is protected, returned as a logical.

Data Types: `logical`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”



Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

---

## Version History

Introduced in R2021b

### See Also

[inlineComponent](#) | [createSimulinkBehavior](#) | [createArchitectureModel](#) | [createStateflowChartBehavior](#) | [extractArchitectureFromSimulink](#) | [linkToModel](#) | [isReference](#) | [Reference Component](#)

### Topics

["Implement Component Behavior Using Simulink"](#)  
["Decompose and Reuse Components"](#)  
["Implement Component Behavior Using Stateflow Charts"](#)  
["Create Simulink Subsystem Behavior Using Subsystem Component"](#)  
["Simulate and Deploy Software Architectures"](#)

## isReference

**Package:** systemcomposer.arch

Find if component is referenced to another model

### Syntax

```
flag = isReference(compObj)
```

### Description

`flag = isReference(compObj)` returns whether or not the component is a reference to another model.

### Examples

#### Find If Component Is Reference

Find whether or not the component is a reference to another model.

This component is not a reference.

```
model = systemcomposer.createModel("archModel",true);
rootArch = get(model,"Architecture");
newComponent = addComponent(rootArch,"newComponent");
flag = isReference(newComponent)
```

```
flag =
    logical
    0
```

This component is a reference.

```
model = systemcomposer.createModel("archModel",true);
rootArch = get(model,"Architecture");
newComponent = addComponent(rootArch,"newComponent");
createSimulinkBehavior(newComponent,"newModel");
flag = isReference(newComponent)
```

```
flag =
    logical
    1
```

### Input Arguments

#### **compObj** — Component

component object | variant component object

Component, specified as a `systemcomposer.arch.Component` or `systemcomposer.arch.VariantComponent` object.

## Output Arguments

### flag — Whether component is reference

true or 1 | false or 0

Whether component is reference, returned as a logical.

Data Types: `logical`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## **Version History**

**Introduced in R2019a**

### **See Also**

`inlineComponent` | `createSimulinkBehavior` | `createArchitectureModel` |  
`createStateflowChartBehavior` | `extractArchitectureFromSimulink` | `linkToModel` |  
Reference Component

### **Topics**

“Implement Component Behavior Using Simulink”  
“Decompose and Reuse Components”  
“Implement Component Behavior Using Stateflow Charts”  
“Create Simulink Subsystem Behavior Using Subsystem Component”  
“Simulate and Deploy Software Architectures”



# IsStereotypeDerivedFrom

**Package:** systemcomposer.query

Create query to select stereotype derived from qualified name

## Syntax

```
query = IsStereotypeDerivedFrom(name)
```

## Description

query = IsStereotypeDerivedFrom(name) creates a query query that the find and createView functions use to select a stereotype from the qualified name name.

## Examples

### Construct Query to Select All Hardware Components

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
scKeylessEntrySystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query for all the hardware components and run the query, displaying one of them.

```
constraint = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent"));
hwComp = find(model, constraint, Recurse=true, IncludeReferenceModels=true);
comp = hwComp(16)
```

```
comp = 1x1 cell array
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Sensor'}
```

## Input Arguments

**name — Stereotype name**

character vector | string

Stereotype name, specified in the form "<profile>.<stereotype>".

Example: "AutoProfile.BaseComponent"

Data Types: char | string

## Output Arguments

### query — Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `HasStereotype` | `getQualifiedName`

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”

## iterate

**Package:** systemcomposer.arch

Iterate over model elements

### Syntax

```
iterate(arch,iterType,iterFunction)
iterate( ____,Name,Value)
iterate( ____,Name,Value,additionalArgs)
```

### Description

`iterate(arch,iterType,iterFunction)` iterates over components in the architecture `arch` in the order specified by `iterType` and invokes the function specified by the function handle `iterFunction` on each component.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

`iterate( ____,Name,Value)` iterates over components in the architecture using all previous syntaxes with additional options.

`iterate( ____,Name,Value,additionalArgs)` iterates over components in the architecture with additional options and passes all trailing arguments, specified as `additionalArgs`, as arguments to `iterFunction`. Name-value arguments with `additionalArgs` must be specified as comma-separated name-value pairs.

### Examples

#### Compute Battery Capacity

For more information on the battery sizing example, see “Battery Sizing and Automotive Electrical System Analysis”.

```
openExample("systemcomposer/BatterySizingAndAutomotiveAnalysisExample")
archModel = systemcomposer.openModel("scExampleAutomotiveElectricalSystemAnalysis");
% Instantiate battery sizing class used by analysis function to store
% analysis results.
objcomputeBatterySizing = computeBatterySizing;
% Run the analysis using the iterator
iterate(archModel,"TopDown",@computeLoad,"Recurse",true,objcomputeBatterySizing);
```

### Input Arguments

**arch — Architecture over which to iterate**

architecture object | architecture instance object

Architecture over which to iterate, specified as an `systemcomposer.arch.Architecture` or `systemcomposer.analysis.ArchitectureInstance` object.

### **iterType — Iteration type**

"PreOrder" | "PostOrder" | "TopDown" | "BottomUp"

Iteration type, specified as "PreOrder", "PostOrder", "TopDown", or "BottomUp".

- **Pre-order** — Start from the top level, move to a child component, and process the subcomponents of that component recursively before moving to a sibling component.
- **Top-Down** — Like pre-order, but process all sibling components before moving to their subcomponents.
- **Post-order** — Start from components with no subcomponents, process each sibling, and then move to parent.
- **Bottom-up** — Like post-order, but process all subcomponents at the same depth before moving to their parents.

Data Types: `char` | `string`

### **iterFunction — Iteration function**

function handle

Iteration function, specified as a function handle to be iterated on each component.

### **additionalArgs — Additional function arguments**

comma-separated list of function arguments

Additional function arguments, specified as a comma-separated list of arguments to be passed to `iterFunction`.

### **Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example:

```
iterate(archModel, "TopDown", @computeLoad, "Recurse", true, objcomputeBatterySizing)
```

### **Recurse — Option to recursively iterate through model components**

true or 1 (default) | false or 0

Option to recursively iterate through model components, specified as a logical 1 (`true`) to recursively iterate or 0 (`false`) to iterate over components only in this architecture and not navigate into the architectures of child components.

Recurse only applies to `systemcomposer.arch.Architecture` objects.

Data Types: `logical`

### **IncludePorts — Option to iterate over components and architecture ports**

false or 0 (default) | true or 1

Option to iterate over components and architecture ports, specified as a logical 0 (`false`) to only iterate over components or 1 (`true`) to iterate over components and architecture ports.

Data Types: `logical`

### **IncludeConnectors — Option to iterate over components and connectors**

`false` or 0 (default) | `true` or 1

Option to iterate over components and connectors, specified as a logical 0 (`false`) to only iterate over components or 1 (`true`) to iterate over components and connectors.

`IncludeConnectors` only applies to `systemcomposer.analysis.ArchitectureInstance` objects.

Data Types: `logical`

### **FollowConnectivity — Option to ensure iteration order**

`false` or 0 (default) | `true` or 1

Option to ensure iteration order according to how components are connected from source to destination, specified as a logical 0 (`false`) or 1 (`true`). If this option is specified as 1 (`true`), iteration type `iterType` has to be either "TopDown" or "BottomUp". If any other option is specified, the iteration type defaults to "TopDown".

`FollowConnectivity` only applies to `systemcomposer.arch.Architecture` objects.

Data Types: `logical`

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>• “Analyze Architecture Model with Analysis Function”</li> <li>• “Analyze Architecture”</li> <li>• “Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>• “Analysis Function Constructs”</li> <li>• “Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

## Version History

Introduced in R2019a

### See Also

`instantiate` | `lookup` | `systemcomposer.analysis.Instance`

### Topics

“Analyze Architecture”



# linkDictionary

**Package:** systemcomposer.arch

Link data dictionary to architecture model

## Syntax

```
linkDictionary(model,dictionaryFile)
```

## Description

`linkDictionary(model,dictionaryFile)` associates the specified Simulink data dictionary with the model. The model cannot have locally defined interfaces.

## Examples

### Link Data Dictionary

Link a data dictionary to a model.

```
model = systemcomposer.createModel("newModel",true);  
dictionary = systemcomposer.createDictionary("newDictionary.slidd");  
linkDictionary(model,"newDictionary.slidd");  
save(dictionary);  
save(model);
```

## Input Arguments

### **model** — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

### **dictionaryFile** — Dictionary file name

character vector | string

Dictionary file name with the `.slidd` extension, specified as a character vector or string. If a dictionary with this name does not exist, one will be created.

Example: "dict\_name.slidd"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• "Create Architecture Model with Interfaces and Requirement Links"</li> <li>• "Define Port Interfaces Between Components"</li> </ul>

Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

[createDictionary](#) | [saveToDictionary](#) | [unlinkDictionary](#) | [openDictionary](#) | [addReference](#) | [removeReference](#)

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## linkToModel

**Package:** systemcomposer.arch

Link component to model

### Syntax

```
modelHandle = linkToModel(component,modelName)
modelHandle = linkToModel(component,modelFileName)
```

### Description

`modelHandle = linkToModel(component,modelName)` links from the component to a model or subsystem.

`modelHandle = linkToModel(component,modelFileName)` links from the component to a model or subsystem defined by its full file name with an SLX or SLXP extension.

### Examples

#### Reuse Component

Save the component named `robotComp` in the architecture model `Robot.slx` and reference it from another component named `electricComp` so that the component `electricComp` uses the architecture of the component `robotComp`.

Create a model `archModel.slx`.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
```

Add two components to the model with the names `electricComp` and `robotComp`.

```
names = ["electricComp","robotComp"];
comp = addComponent(arch,names);
```

Save `robotComp` in the `Robot.slx` model so the component references the model.

```
saveAsModel(comp(2),"Robot");
```

Link the `electricComp` component to the same model `Robot.slx` so it uses the architecture of the original `robotComp` component and references the architecture model `Robot.slx`.

```
linkToModel(comp(1),"Robot");
```

Clean up the model.

```
Simulink.BlockDiagram.arrangeSystem("archModel");
```

## Input Arguments

### **component** — Component

component object

Component with no sub-components, specified as a `systemcomposer.arch.Component` object.

### **modelName** — Model or subsystem name

character vector | string

Model or subsystem name for an existing model or subsystem that defines the architecture or behavior of the component, specified as a character vector or string. Models or subsystems of the same name prioritize protected models with the SLXP extension.

Example: "Robot"

Data Types: char | string

### **modelName** — Model or subsystem file name

character vector | string

Model or subsystem file name for an existing model or subsystem that defines the architecture or behavior of the component, specified as a character vector or string.

Example: "Model.slx"

Example: "ProtectedModel.slxp"

Data Types: char | string

## Output Arguments

### **modelHandle** — Handle to linked model or subsystem

numeric value

Handle to linked model or subsystem, returned as a numeric value.

Data Types: double

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”



Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>

Term	Definition	Application	More Information
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>“Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>“Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2019a

### See Also

`inlineComponent` | `createSimulinkBehavior` | `createArchitectureModel` | `createStateflowChartBehavior` | `extractArchitectureFromSimulink` | `isReference` | Reference Component

### Topics

“Implement Component Behavior Using Simulink”  
 “Decompose and Reuse Components”  
 “Implement Component Behavior Using Stateflow Charts”  
 “Create Simulink Subsystem Behavior Using Subsystem Component”  
 “Simulate and Deploy Software Architectures”

# systemcomposer.allocation.load

Load allocation set

## Syntax

```
allocSet = systemcomposer.allocation.load(name)
```

## Description

`allocSet = systemcomposer.allocation.load(name)` loads the allocation set with the given name, if it exists on the MATLAB path.

## Examples

### Load Allocation Set and Open in Allocation Editor

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario,sourceComp,targetComp);
```

Save the allocation set.

```
save(allocSet)
```

Close the allocation set.

```
close(allocSet)
```

Load the allocation set `MyNewAllocation.mldatx`.

```
allocSet = systemcomposer.allocation.load("MyNewAllocation");
```

Open the **Allocation Editor**.

systemcomposer.allocation.editor

## Input Arguments

### name — Name of allocation set

character vector | string

Name of allocation set, specified as a character vector or string.

Example: "MyNewAllocation"

Data Types: char | string

## Output Arguments

### allocSet — Allocation set

allocation set object

Allocation set, returned as a systemcomposer.allocation.AllocationSet object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called Scenario 1.	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

## **See Also**

`createAllocationSet` | `open` | `closeAll`

## **Topics**

“Create and Manage Allocations Programmatically”

## systemcomposer.profile.Profile.load

Load profile from file

### Syntax

```
profile = systemcomposer.profile.Profile.load(profileName)
```

### Description

`profile = systemcomposer.profile.Profile.load(profileName)` loads a profile from a file name.

### Examples

#### Load Profile

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Load the profile into another variable.

```
newProfile = systemcomposer.profile.Profile.load("LatencyProfile")
newProfile =

    Profile with properties:

        Name: 'LatencyProfile'
    FriendlyName: ''
    Description: ''
    Stereotypes: [1x5 systemcomposer.profile.Stereotype]
```

### Input Arguments

**profileName** — Name of profile

character vector | string

Name of profile, specified as a character vector or string. Profile must be available on the MATLAB path with a `.xml` extension.

Example: "LatencyProfile"

Data Types: char | string

## Output Arguments

### **profile** – Profile

profile object

Profile, returned as a `systemcomposer.profile.Profile` object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• "Set Properties"</li> <li>• "Add Properties with Stereotypes"</li> <li>• "Set Properties for Analysis"</li> </ul>

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"><li>• “Define Profiles and Stereotypes”</li><li>• “Use Stereotypes and Profiles”</li></ul>

## **Version History**

**Introduced in R2019a**

### **See Also**

systemcomposer.profile.Profile | open | editor | save | find | closeAll | close | createProfile

### **Topics**

“Define Profiles and Stereotypes”



# systemcomposer.analysis.loadInstance

Load architecture instance

## Syntax

```
instance = systemcomposer.analysis.loadInstance(fileName, overwrite)
```

## Description

`instance = systemcomposer.analysis.loadInstance(fileName, overwrite)` loads an architecture instance from a MAT-file.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

## Examples

### Load Architecture Instance from MAT-File

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency", ...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure", Type="boolean");
connLatency.addProperty("linkDistance", Type="double");

nodeLatency = profile.addStereotype("NodeLatency", ...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");

portLatency = profile.addStereotype("PortLatency", ...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth", Type="double");
portLatency.addProperty("dummy", Type="int32");

profile.save
```

Instantiate all stereotypes in the profile.

```
model = systemcomposer.createModel("archModel", true);
instance = instantiate(model.Architecture, "LatencyProfile", "NewInstance");
```

Save the architecture instance.

```
instance.save("InstanceFile");
```

Delete the architecture instance.

```
systemcomposer.analysis.deleteInstance(instance);
```

Load the architecture instance.

```
loadedInstance = systemcomposer.analysis.loadInstance("InstanceFile");
```

## Input Arguments

### **fileName** — MAT-file that contains architecture instance

character vector | string

MAT-file that contains architecture instance, specified as a character vector or string.

Data Types: char | string

### **overwrite** — Whether to overwrite instance if it already exists in workspace

true or 1 | false or 0

Whether to overwrite instance if it already exists in workspace, specified as a logical 1 (true) so the load operation overwrites duplicate instances in the workspace or 0 (false) if not.

## Output Arguments

### **instance** — Loaded architecture instance

architecture instance object

Loaded architecture instance, returned as a `systemcomposer.analysis.ArchitectureInstance` object.

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>

Term	Definition	Application	More Information
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	"Run Analysis Function"
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	"Create a Model Instance for Analysis"

## Version History

Introduced in R2019a

### See Also

instantiate | systemcomposer.analysis.Instance | deleteInstance | save | refresh | update

### Topics

"Write Analysis Function"

## systemcomposer.loadModel

Load System Composer model

### Syntax

```
model = systemcomposer.loadModel(modelName)
```

### Description

`model = systemcomposer.loadModel(modelName)` loads the architecture model with name `modelName` and returns the `systemcomposer.arch.Model` object. The loaded model is not displayed. The architecture model must exist on the MATLAB path.

### Examples

#### Load Model

Create, save, and load a model. Display the model's properties.

```
model = systemcomposer.createModel("new_arch", true);
model.save;
loadedModel = systemcomposer.loadModel("new_arch")

loadedModel =

    model with properties:

        Name: 'new_arch'
        Architecture: [1x1 systemcomposer.arch.Architecture]
        SimulinkHandle: 2.0005
        Views: [0x0 systemcomposer.view.ViewArchitecture]
        Profiles: [0x0 systemcomposer.profile.Profile]
        InterfaceDictionary: [1x1 systemcomposer.interface.Dictionary]
```

### Input Arguments

#### **modelName** — Name of model

character vector | string

Name of model, specified as a character vector or string.

Example: "exMobileRobot"

Data Types: char | string

### Output Arguments

#### **model** — Architecture model

model object

Architecture model, returned as a `systemcomposer.arch.Model` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

save | open | systemcomposer.createModel

### Topics

"Create Architecture Model"

# systemcomposer.loadProfile

Load profile by name

## Syntax

```
profile = systemcomposer.loadProfile(profileName)
```

## Description

`profile = systemcomposer.loadProfile(profileName)` loads a profile with the specified file name.

## Examples

### Load Profile

Create a model.

```
model = systemcomposer.createModel("archModel",true);
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

Save the profile and load the profile. In this example, `profileNew` is equal to `profile`.

```
save(profile);
profileNew = systemcomposer.loadProfile("LatencyProfile");
```

## Input Arguments

### **profileName** — Name of profile

character vector | string

Name of profile, specified as a character vector or string. Profile must be available on the MATLAB path with a `.xml` extension.

Example: "LatencyProfile"

Data Types: char | string

## Output Arguments

### **profile** — Profile

profile object

Profile, returned as a `systemcomposer.profile.Profile` object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a



## **See Also**

[applyProfile](#) | [createProfile](#) | [editor](#) | [systemcomposer.profile.Profile](#)

## **Topics**

[“Define Profiles and Stereotypes”](#)

# lookup

**Package:** systemcomposer.arch

Search for architectural element

## Syntax

```
element = lookup(object,Name,Value)
instance = lookup(object,Name,Value)
```

## Description

`element = lookup(object,Name,Value)` finds an architectural element based on its universal unique identifier (UUID) or full path.

`instance = lookup(object,Name,Value)` finds an architectural element instance based on its universal unique identifier (UUID) or full path.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

## Examples

### Look Up Component by Path

This example shows how to find a component by path in a robot model.

```
arch = systemcomposer.loadModel("Robot");
component = lookup(arch,Path="Robot/Sensor")
```

```
component =
  Component with properties:

    IsAdapterComponent: 0
      Architecture: [1x1 systemcomposer.arch.Architecture]
        Name: 'Sensor'
        Parent: [1x1 systemcomposer.arch.Architecture]
        Ports: [1x2 systemcomposer.arch.ComponentPort]
        OwnedPorts: [1x2 systemcomposer.arch.ComponentPort]
      OwnedArchitecture: [1x1 systemcomposer.arch.Architecture]
      Parameters: [0x0 systemcomposer.arch.Parameter]
      Position: [349 74 469 174]
      Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 10.0021
    SimulinkModelHandle: 0.0023
      UUID: 'cfd62628-d365-47e4-8492-62cfeaa8dc15'
      ExternalUID: ''
```

## Input Arguments

### **object** — Architecture model or instance object

model object | architecture instance object

Architecture model or instance object to look up, specified as a `systemcomposer.arch.Model` or `systemcomposer.analysis.ArchitectureInstance` object.

### **Name-Value Arguments**

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

Example: `lookup(arch,Path="Robot/Sensor")`

### **UUID** — UUID

character vector | string

UUID to use for search, specified as a character vector or string of the UUID.

Example: `lookup(arch,UUID="f43c9d51-9dc6-43fc-b3af-95d458b81248")`

Data Types: `char` | `string`

### **SimulinkHandle** — Simulink handle

double

Simulink handle to use for search, specified as the `SimulinkHandle` value.

Example: `lookup(arch,SimulinkHandle=9.0002)`

Data Types: `double`

### **Path** — Full path

character vector | string

Full path, specified as a character vector or string.

Example: `lookup(arch,Path="Robot/Sensor")`

Data Types: `char` | `string`

## Output Arguments

### **element** — Model element

architecture object | component object | port object | connector object | physical connector object | data interface object | value type object | physical interface object

Model element, returned as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`,

`systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, or `systemcomposer.interface.PhysicalInterface` object.

### instance – Element instance

component instance | port instance | connector instance

Element instance, returned as a `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>• "Analyze Architecture Model with Analysis Function"</li> <li>• "Analyze Architecture"</li> <li>• "Simple Roll-Up Analysis Using Robot System with Properties"</li> </ul>

Term	Definition	Application	More Information
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>• “Analysis Function Constructs”</li> <li>• “Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

## Version History

Introduced in R2019a

### See Also

find | createView | getQualifiedName | systemcomposer.view.ElementGroup | systemcomposer.analysis.Instance | iterate | instantiate

### Topics

“Analyze Architecture”

“Create Architectural Views Programmatically”

# makeOwnedInterfaceShared

**Package:** systemcomposer.arch

Convert owned interface to shared interface

## Syntax

```
makeOwnedInterfaceShared(archPort, newInterfaceName)
```

## Description

`makeOwnedInterfaceShared(archPort, newInterfaceName)` converts an owned interface on the port `archPort` into a shared interface with name `newInterfaceName` in the interface data dictionary used in the architecture model.

## Examples

### Make Owned Interface Shared

Create an architecture port on a component in an architecture model.

```
modelName = "archModel";
model = systemcomposer.createModel(modelName, true);
comp = model.Architecture.addComponent("Component1");
inport = comp.Architecture.addPort("InBus", "in");
```

Add a shared interface to the model.

```
interfaceDict = model.InterfaceDictionary;
SharedInterface = interfaceDict.addInterface("SharedInterface");
SharedInterface.addElement("SharedElem_X");
SharedInterface.addElement("SharedElem_Y");
```

Create an owned interface on the architecture port.

```
ownedInterface = inport.createInterface("DataInterface");
ownedInterface.removeElement("elem0");
elemA = ownedInterface.addElement("A");
ownedInterface.addElement("B", DataType="single", Dimensions="1", ...
Units="m", Complexity="real", Maximum="200", Minimum="0", ...
Description="Length value");
```

Convert the owned interface to a shared interface.

```
convertedInterface = inport.makeOwnedInterfaceShared("convertedInterface")
```

```
convertedInterface =
```

DataInterface with properties:

```
Owner: [1x1 systemcomposer.interface.Dictionary]
Name: 'convertedInterface'
Elements: [1x2 systemcomposer.interface.DataElement]
Model: [1x1 systemcomposer.arch.Model]
```

```

        UUID: '59a41ae1-e04d-479c-81e6-881230bad662'
    ExternalUID: ''

```

## Input Arguments

### archPort — Architecture port

architecture port object

Architecture port, specified as a `systemcomposer.arch.ArchitecturePort` object.

### newInterfaceName — New interface name

character vector | string

New interface name, specified as a character vector or string.

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>“Manage Interfaces with Data Dictionaries”</li> <li>“Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>“Create Architecture Model with Interfaces and Requirement Links”</li> <li>“Define Port Interfaces Between Components”</li> </ul>



Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2022a

### See Also

`createModel` | `createInterface` | `addElement` | `addInterface` | `addValueType`

### Topics

“Define Port Interfaces Between Components”

“Assign Interfaces to Ports”

“Manage Interfaces with Data Dictionaries”

# makeVariant

**Package:** systemcomposer.arch

Convert component to variant choice

## Syntax

```
[variantComp,choices] = makeVariant(component)
[variantComp,choices] = makeVariant(component,Name,Value)
```

## Description

`[variantComp,choices] = makeVariant(component)` converts the component `component` to a variant choice component and returns the parent Variant Component block object `variantComp` and available variant choice components `choices`.

`[variantComp,choices] = makeVariant(component,Name,Value)` converts the component `component` to a variant choice component with additional options and returns the parent Variant Component block object `variantComp` and available variant choice components `choices`.

## Examples

### Make Variant Component

Create a top-level architecture model.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName,true);
rootArch = get(arch,"Architecture");
```

Create a new component.

```
newComponent = addComponent(rootArch,"Component");
```

Add ports to the component.

```
inPort = addPort(newComponent.Architecture,"testSig","in");
outPort = addPort(newComponent.Architecture,"testSig","out");
```

Make the component into a variant component.

```
[variantComp,choices] = makeVariant(newComponent)
```

```
variantComp =
```

```
  VariantComponent with properties:
```

```
    Architecture: [1x1 systemcomposer.arch.Architecture]
      Name: 'Component'
    Parent: [1x1 systemcomposer.arch.Architecture]
    Ports: [1x2 systemcomposer.arch.ComponentPort]
    OwnedPorts: [1x2 systemcomposer.arch.ComponentPort]
```

```

OwnedArchitecture: [1x1 systemcomposer.arch.Architecture]
  Parameters: [0x0 systemcomposer.arch.Parameter]
    Position: [15 15 65 83]
      Model: [1x1 systemcomposer.arch.Model]
SimulinkHandle: 207.0020
SimulinkModelHandle: 0.0024
  UUID: 'b32a5b3d-3493-4f3f-baab-f9d99f866a41'
  ExternalUID: ''

```

```
choices =
```

```
  Component with properties:
```

```

  IsAdapterComponent: 0
    Architecture: [1x1 systemcomposer.arch.Architecture]
      Name: 'Component'
      Parent: [1x1 systemcomposer.arch.Architecture]
      Ports: [1x2 systemcomposer.arch.ComponentPort]
      OwnedPorts: [1x2 systemcomposer.arch.ComponentPort]
OwnedArchitecture: [1x1 systemcomposer.arch.Architecture]
  Parameters: [0x0 systemcomposer.arch.Parameter]
    Position: [50 20 100 80]
      Model: [1x1 systemcomposer.arch.Model]
SimulinkHandle: 2.0188
SimulinkModelHandle: 0.0024
  UUID: '352b8000-7881-41e1-8ec3-0287bc6d94ab'
  ExternalUID: ''

```

## Input Arguments

### component — Component

component object

Component to be converted to variant choice component, specified as a `systemcomposer.arch.Component` object.

### Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1, ..., NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose `Name` in quotes.*

```

Example: [variantComp,choices] =
makeVariant(newComponent,Name="NewVariantComponent",Label="NewVariantChoice",
Choices=["NewVariantChoiceA","NewVariantChoiceB","NewVariantChoiceC"],ChoiceL
abels=["Choice A","Choice B","Choice C"])

```

### Name — Name of variant component

character vector | string

Name of variant component, specified as a character vector or string.

```
Example: [variantComp,choices] =
makeVariant(newComponent,Name="NewVariantComponent")
```

Data Types: char | string

### **Label — Label of variant choice**

character vector | string

Label of variant choice from converted component, specified as a character vector or string.

```
Example: [variantComp,choices] =
makeVariant(newComponent,Name="NewVariantComponent",Label="NewVariantChoice")
```

Data Types: char | string

### **Choices — Variant choice names**

cell array of character vectors | array of strings

Variant choice names, specified as a cell array of character vectors or an array of strings. Additional variant choices are also added to the new variant component, along with the active choice from the converted component.

```
Example: [variantComp,choices] =
makeVariant(newComponent,Choices=["NewVariantChoiceA","NewVariantChoiceB","NewVariantChoiceC"])
```

Data Types: char | string

### **ChoiceLabels — Variant choice labels**

cell array of character vectors | array of strings

Variant choice labels, specified as a cell array of character vectors or an array of strings.

```
Example: [variantComp,choices] =
makeVariant(newComponent,Choices=["NewVariantChoiceA","NewVariantChoiceB","NewVariantChoiceC"],ChoiceLabels=["Choice A","Choice B","Choice C"])
```

Data Types: char | string

## **Output Arguments**

### **variantComp — Variant component**

variant component object

Variant component, returned as a `systemcomposer.arch.VariantComponent` object.

### **choices — Variant choices**

array of component objects

Variant choices, returned as an array of `systemcomposer.arch.Component` objects.

Data Types: char

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li><i>Component ports</i> are interaction points on the component to other components.</li> <li><i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## Version History

Introduced in R2019a

### See Also

[addChoice](#) | [getChoices](#) | [Variant Component](#) | [addVariantComponent](#)

### Topics

"Create Variants"

## modifyQuery

**Package:** `systemcomposer.view`

Modify architecture view query and property groupings

### Syntax

```
modifyQuery(view,select)
modifyQuery(view,select,groupBy)
```

### Description

`modifyQuery(view,select)` modifies the query `select` on the view `view`.

`modifyQuery(view,select,groupBy)` modifies the query `select` on the view `view` and the property based groupings `groupBy`.

### Examples

#### Modify Query and Remove Groupings

Open the keyless entry system example and create a view. Specify the color as light blue, the query as all components, and group by the review status.

```
import systemcomposer.query.*

scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("All Components Grouped by Review Status",...
    Color="lightblue",Select=AnyComponent,...
    GroupBy="AutoProfile.BaseComponent.ReviewStatus");
```

Open the Architecture Views Gallery to see the new view All Components Grouped by Review Status.

```
model.openViews
```

Create a new query for all hardware components. Use the new query to modify the existing query on the view. Remove the property based groupings by passing in an empty cell array `{}`. Observe the change in your view.

```
constraint = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent"));
view.modifyQuery(constraint,{})
```

### Input Arguments

**view** — Architecture view

view object



Architecture view, specified as a `systemcomposer.view.View` object.

### **select – Query**

constraint object

Query to use to populate view, specified as a `systemcomposer.query.Constraint` object.

A constraint can contain a subconstraint that can be joined with another constraint using AND or OR. A constraint can be negated using NOT.

Example:

```
HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent"))
```

### **Query Objects and Conditions for Constraints**

Query Object	Condition
Property	A non-evaluated value for the given property or stereotype property.
PropertyValue	An evaluated property value from a System Composer object or a stereotype property.
HasConnector	A component has a connector that satisfies the given subconstraint.
HasPort	A component has a port that satisfies the given subconstraint.
HasInterface	A port has an interface that satisfies the given subconstraint.
HasInterfaceElement	An interface has an interface element that satisfies the given subconstraint.
HasStereotype	An architecture element has a stereotype that satisfies the given subconstraint.
IsInRange	A property value is within the given range.
AnyComponent	An element is a component and not a port or connector.
IsStereotypeDerivedFrom	A stereotype is derived from the given stereotype.

### **groupBy – Grouping criteria**

cell array of character vectors | empty cell array

Grouping criteria, specified as a cell array of character vectors in the form '`<profile>.<stereotype>.<property>`'. The order of the cell array dictates the order of the grouping. If an empty cell array `{}` is passed into `groupBy`, all the groupings are removed.

Example:

```
{'AutoProfile.MechanicalComponent.mass', 'AutoProfile.MechanicalComponent.cost'}
```

Data Types: char

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

systemcomposer.view.View | createView | getView | deleteView | openViews | runQuery | removeQuery | systemcomposer.view.ElementGroup | getQualifiedName

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

## open

**Package:** systemcomposer.profile

Open profile

### Syntax

```
open(profile)
```

### Description

open(profile) opens a profile in the **Profile Editor**.

### Examples

#### Open Profile

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");  
  
latencybase = profile.addStereotype("LatencyBase");  
latencybase.addProperty("latency",Type="double");  
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");  
  
connLatency = profile.addStereotype("ConnectorLatency",...  
Parent="LatencyProfile.LatencyBase");  
connLatency.addProperty("secure",Type="boolean");  
connLatency.addProperty("linkDistance",Type="double");  
  
nodeLatency = profile.addStereotype("NodeLatency",...  
Parent="LatencyProfile.LatencyBase");  
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");  
  
portLatency = profile.addStereotype("PortLatency",...  
Parent="LatencyProfile.LatencyBase");  
portLatency.addProperty("queueDepth",Type="double");  
portLatency.addProperty("dummy",Type="int32");  
  
profile.save
```

Open the profile in the **Profile Editor**.

```
open(profile)
```

### Input Arguments

#### profile — Profile

profile object

Profile, specified as a systemcomposer.profile.Profile object.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

createProfile | find | editor | save | load | close | closeAll

**Topics**

“Define Profiles and Stereotypes”

# systemcomposer.allocation.open

Open allocation set in Allocation Editor

## Syntax

```
allocSet = systemcomposer.allocation.open(name)
```

## Description

`allocSet = systemcomposer.allocation.open(name)` opens allocation set specified by name in the **Allocation Editor**. The allocation set must be on the MATLAB path.

## Examples

### Create and Open Allocation Set

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation",true);
sourceComp = addComponent(get(mSource,"Architecture"),"Source_Component");
mTarget = systemcomposer.createModel("Target_Model_Allocation",true);
targetComp = addComponent(get(mTarget,"Architecture"),"Target_Component");
```

Create the allocation set MyNewAllocation.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation",...
    "Source_Model_Allocation","Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet,"Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario,sourceComp,targetComp);
```

Save the allocation set.

```
save(allocSet)
```

Open the **Allocation Editor** with the allocation set highlighted.

```
systemcomposer.allocation.open(allocSet);
```

## Input Arguments

### name — Name of allocation set

allocation set object | character vector | string

Name of allocation set, specified as an `systemcomposer.allocation.AllocationSet` object, character vector, or string.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

### See Also

`createAllocationSet` | `load`

### Topics

“Create and Manage Allocations Programmatically”



# open

**Package:** `systemcomposer.arch`

Open architecture model

## Syntax

```
open(model)
```

## Description

`open(model)` opens the specified model in System Composer.

## Examples

### Create and Open Model

```
model = systemcomposer.createModel("modelName");
open(model)
```

## Input Arguments

**model** — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

## **Version History**

Introduced in R2019a

### **See Also**

`createModel` | `openModel`

### **Topics**

“Create Architecture Model”

## systemcomposer.openDictionary

Open data dictionary

### Syntax

```
dictionary = systemcomposer.openDictionary(dictionaryName)
```

### Description

`dictionary = systemcomposer.openDictionary(dictionaryName)` opens an existing Simulink data dictionary to hold interfaces and returns the `systemcomposer.interface.Dictionary` object.

### Examples

#### Open Existing Dictionary

Create a dictionary and open the dictionary.

```
systemcomposer.createDictionary("my_dictionary.sldd");  
dictionary = systemcomposer.openDictionary("my_dictionary.sldd");
```

### Input Arguments

#### **dictionaryName** — Name of existing data dictionary

character vector | string

Name of existing data dictionary, specified as a character vector or string. The name must include the `.sldd` extension.

Example: "my\_dictionary.sldd"

Data Types: char | string

### Output Arguments

#### **dictionary** — Dictionary

dictionary object

Dictionary, returned as a `systemcomposer.interface.Dictionary` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>

Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

`linkDictionary` | `saveToDictionary` | `unlinkDictionary` | `createDictionary` | `addReference` | `removeReference`

### Topics

“Define Port Interfaces Between Components”

“Manage Interfaces with Data Dictionaries”



# systemcomposer.openModel

Open System Composer model

## Syntax

```
model = systemcomposer.openModel(modelName)
```

## Description

`model = systemcomposer.openModel(modelName)` opens the architecture model with name `modelName` for editing and returns the `systemcomposer.arch.Model` object. The model must exist on the MATLAB path.

## Examples

### Open Model

Create, save, and close a model. Open the model and display the model's properties.

```
model = systemcomposer.createModel("new_arch");
model.close;
model.save;
openedModel = systemcomposer.openModel("new_arch")

openedModel =

    model with properties:

        Name: 'new_arch'
        Architecture: [1x1 systemcomposer.arch.Architecture]
        SimulinkHandle: 2.0005
        Views: [0x0 systemcomposer.view.ViewArchitecture]
        Profiles: [0x0 systemcomposer.profile.Profile]
        InterfaceDictionary: [1x1 systemcomposer.interface.Dictionary]
```

## Input Arguments

### modelName — Name of model

character vector | string

Name of model, specified as a character vector or string.

Example: "exMobileRobot"

Data Types: char | string

## Output Arguments

### model — Architecture model

model object

Architecture model, returned as a `systemcomposer.arch.Model` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

[open](#) | [close](#)

### Topics

"Create Architecture Model"

## openViews

**Package:** systemcomposer.arch

Open Architecture Views Gallery

### Syntax

```
openViews(model)
```

### Description

`openViews(model)` opens the **Architecture Views Gallery** for the specified model, `model`. If the model is already open, `openViews` will bring the views to the front.

### Examples

#### Open Views Editor

Open the keyless entry system example and create a view. Open the Architecture Views Gallery for the model.

```
scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
fobSupplierView = model.createView("FOB Locator System Supplier Breakdown",...
    Color="lightblue");
openViews(model)
```

### Input Arguments

#### **model** — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”

Term	Definition	Application	More Information
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `systemcomposer.view.ElementGroup`

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

## Property

**Package:** `systemcomposer.query`

Create query to select non-evaluated values for object properties or stereotype properties for elements

### Syntax

```
query = Property(name)
```

### Description

`query = Property(name)` creates a query `query` that the `find` and `createView` functions use to select non-evaluated values for object properties or stereotype properties for elements based on a specified property name `name`.

### Examples

#### Find Model Elements that Satisfy Property

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
scKeylessEntrySystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query to find components that contain `Sensor` in their `Name` property and run the query, displaying the first.

```
constraint = contains(Property("Name"), "Sensor");
sensors = find(model, constraint, Recurse=true, IncludeReferenceModels=true);
query = sensors(1)
```

```
query = 1x1 cell array
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Sensor/Door Lock Sen
```

### Input Arguments

**name** — Property name

character vector | string

Property name for model element, specified in the form "`<profile>.<stereotype>.<property>`" or any property on the designated class.



Example: "Name"

Example: "AutoProfile.BaseComponent.Latency"

Data Types: char

## Output Arguments

### query – Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>"Create Architecture Views Interactively"</li> <li>"Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2019b

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `PropertyValue` | `getQualifiedName`

### Topics

“Create Architectural Views Programmatically”

“Modeling System Architecture of Keyless Entry System”

# Property Value

**Package:** `systemcomposer.query`

Create query to select property from object or stereotype property and then evaluate property value

## Syntax

```
query = PropertyValue(name)
```

## Description

`query = PropertyValue(name)` creates a query `query` that the `find` and `createView` functions use to select object properties or stereotype properties for elements based on specified property name `name` and then evaluate the property value.

## Examples

### Find Model Elements that Satisfy Property Value

Import the package that contains all of the System Composer™ queries.

```
import systemcomposer.query.*
```

Open the Simulink® project file for the keyless entry system.

```
sckeylessentrysystem
```

Load the architecture model.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Create a query to find components with a Latency property value of 30 and run the query.

```
constraint = PropertyValue("AutoProfile.BaseComponent.Latency")==30;
```

```
latency = find(model,constraint,Recurse=true,IncludeReferenceModels=true)
```

```
latency = 4x1 cell
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Actuator' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Actuator' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Actuator' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Actuator' }
```

## Input Arguments

**name — Property name**

character vector | string

Property name for model element, specified in the form "`<profile>.<stereotype>.<property>`" or any property on the designated class.

Example: "Name"

Example: "AutoProfile.BaseComponent.Latency"

Data Types: char

## Output Arguments

### query – Query

query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>"Create Architecture Views Interactively"</li> <li>"Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	"Display Component Hierarchy and Architecture Hierarchy Using Views"

## Version History

Introduced in R2019b

### See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `Property` | `getQualifiedName`

### Topics

"Create Architectural Views Programmatically"

"Modeling System Architecture of Keyless Entry System"

# refresh

**Package:** systemcomposer.analysis

Refresh architecture instance

## Syntax

```
refresh(instance)
```

## Description

`refresh(instance)` refreshes an architecture instance `instance` to mirror the changes in the specification model. The `refresh` method is part of the `systemcomposer.analysis.ArchitectureInstance` class.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

## Examples

### Refresh Architecture Instance

Refresh an architecture instance to mirror the changes in the specification model.

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Instantiate all stereotypes in a profile.

```
model = systemcomposer.createModel("archModel",true);
instance = instantiate(model.Architecture,"LatencyProfile","NewInstance");
```

Apply the profile to the model. Apply the stereotype to the architecture.

```
model.applyProfile("LatencyProfile");
model.Architecture.applyStereotype("LatencyProfile.LatencyBase");
```

Refresh the architecture instance according to the specification model. Get the default value for the "dataRate" property on the architecture instance.

```
instance.refresh;
value = instance.getValue("LatencyProfile.LatencyBase.dataRate")
```

```
value =
```

```
10
```

## Input Arguments

### instance — Architecture instance

architecture instance object

Architecture instance to be refreshed, specified as a `systemcomposer.analysis.ArchitectureInstance` object.

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>

Term	Definition	Application	More Information
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

## Version History

Introduced in R2019a

### See Also

`instantiate` | `systemcomposer.analysis.Instance` | `loadInstance` | `deleteInstance` | `update` | `save` | `lookup` | `iterate`

### Topics

“Write Analysis Function”



# removeComponent

**Package:** systemcomposer.view

(Removed) Remove component from view

---

**Note** The removeComponent function has been removed. You can create a view using the createView function with a selection query, remove the query using the removeQuery function, and remove a component using the removeElement function. For further details, see “Compatibility Considerations”.

---

## Syntax

```
removeComponent(object, compPath)
```

## Description

removeComponent(object, compPath) removes the component with the specified path.

removeComponent is a method from the class systemcomposer.view.ViewArchitecture.

## Examples

### Remove Component from View

Create a model, extract its architecture, and add three components.

```
model = systemcomposer.createModel('mobileRobotAPI');
arch = model.Architecture;
components = addComponent(arch,{'Sensor', 'Planning', 'Motion'});
```

Create a view architecture, a view component, and add a component. Open the **Architecture Views Gallery** to view the component.

```
view = model.createViewArchitecture('NewView');
viewComp = fobSupplierView.createViewComponent('ViewComp');
viewComp.Architecture.addComponent('mobileRobotAPI/Motion');
openViews(model);
```

Remove the component from the view and check the **Architecture Views Gallery**.

```
viewComp.Architecture.removeComponent('mobileRobotAPI/Motion');
```

## Input Arguments

### object – View architecture

view architecture object

View architecture, specified as a systemcomposer.view.ViewArchitecture object.

**compPath — Path to component**

character vector

Path to component, including the name of the top-level model, specified as a character vector.

Data Types: char

## Version History

**Introduced in R2019b****R2021a: removeComponent function has been removed***Errors starting in R2021a*

The `removeComponent` function is removed in R2021a with the introduction of new views APIs. For more information on how to create and edit a view programmatically, see “Create Architectural Views Programmatically”.

**See Also**

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `systemcomposer.view.ElementGroup`

**Topics**

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# removeElement

**Package:** systemcomposer.interface

Remove element

## Syntax

```
removeElement(interface, name)
```

## Description

removeElement(interface, name) removes an element with name name from an interface interface.

## Examples

### Remove Data Element from Data Interface

Add a data interface newInterface to the interface dictionary of the model. Add a data element newElement with data type double to the data interface, then remove the data element.

```
arch = systemcomposer.createModel("newModel", true);
interface = addInterface(arch.InterfaceDictionary, "newInterface");
element = addElement(interface, "newElement", DataType="double");
removeElement(interface, "newElement")
```

### Remove Physical Element from Physical Interface

Add a physical interface newPhysicalInterface to the interface dictionary of the model. Add a physical element newElement with domain type electrical.electrical to the physical interface, then remove the physical element.

```
arch = systemcomposer.createModel("newModel", true);
interface = addPhysicalInterface(arch.InterfaceDictionary, "newPhysicalInterface");
element = addElement(interface, "newElement", Type="electrical.electrical");
removeElement(interface, "newElement")
```

## Input Arguments

### interface – Interface

data interface object | physical interface object | service interface object

Interface, specified as a systemcomposer.interface.DataInterface, systemcomposer.interface.PhysicalInterface, or systemcomposer.interface.ServiceInterface object.

### name – Element name

character vector | string

Element name, specified as a character vector or string. An element name must be a valid MATLAB variable name.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>“Manage Interfaces with Data Dictionaries”</li> <li>“Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>“Create Architecture Model with Interfaces and Requirement Links”</li> <li>“Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>Pins or wires in a connector or harness.</li> <li>Messages transmitted across a bus.</li> <li>Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>“Create Interfaces”</li> <li>“Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	"Create Value Types as Interfaces"
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	"Define Owned Interfaces Local to Ports"
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the "Interface Adapter" dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• "Interface Adapter"</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

**See Also**

addElement | createDictionary | getElement | getInterfaceNames | getInterface | linkDictionary | getSourceElement | getDestinationElement | unlinkDictionary

**Topics**

“Specify Physical Interfaces on Ports”  
“Create Interfaces”  
“Manage Interfaces with Data Dictionaries”

# removeElement

**Package:** systemcomposer.view

Remove component from element group of view

## Syntax

```
removeElement(elementGroup, component)
```

## Description

`removeElement(elementGroup, component)` adds the component `component` to the element group `elementGroup` of an architecture view.

---

**Note** This function cannot be used when a selection query or grouping is defined on the view. To remove the query, run `removeQuery`.

---

## Examples

### Add Elements to View and Remove Elements from View

Open the keyless entry system example and create a view, `newView`.

```
sKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("newView");
```

Open the Architecture Views Gallery to see `newView`.

```
model.openViews
```

Add an element to the view by path.

```
view.Root.addElement("KeylessEntryArchitecture/Lighting System/Headlights")
```

Add an element to the view by object.

```
component = model.lookup(Path="KeylessEntryArchitecture/Lighting System/Cabin Lights");
view.Root.addElement(component)
```

Remove an element to the view by path.

```
view.Root.removeElement("KeylessEntryArchitecture/Lighting System/Headlights")
```

Remove an element to the view by object.

```
view.Root.removeElement(component)
```

## Input Arguments

### elementGroup — Element group

element group object

Element group for view, specified as a `systemcomposer.view.ElementGroup` object.

### component — Component

component object | variant component object | array of component objects | array of variant component objects | path to component | cell array of component paths

Component to remove from view, specified as a `systemcomposer.arch.Component` object, a `systemcomposer.arch.VariantComponent` object, an array of `systemcomposer.arch.Component` objects, an array of `systemcomposer.arch.VariantComponent` objects, the path to a component, or a cell array of component paths.

Example: "KeylessEntryArchitecture/Lighting System/Headlights"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>"Create Architecture Views Interactively"</li> <li>"Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"



Term	Definition	Application	More Information
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

lookup | openViews | createView | getView | deleteView | systemcomposer.view.ElementGroup | systemcomposer.view.View | addElement | getSubGroup | deleteSubGroup | createSubGroup | getQualifiedName

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

## removeInterface

**Package:** systemcomposer.interface

Remove named interface from interface dictionary

### Syntax

```
removeInterface(dictionary, name)
```

### Description

`removeInterface(dictionary, name)` removes the interface specified by name from the interface dictionary `dictionary`.

### Examples

#### Remove Interface

Create a new model. Add a data interface `newInterface` to the interface dictionary of the model.

```
arch = systemcomposer.createModel("archModel");  
addInterface(arch.InterfaceDictionary, "newInterface");
```

Open the model, then open the **Interface Editor**. Confirm that an interface `newInterface` exists.

```
open(arch)
```

Remove the interface.

```
removeInterface(arch.InterfaceDictionary, "newInterface");
```

View the **Interface Editor**. Confirm that `newInterface` is removed.

### Input Arguments

#### **dictionary** — Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

#### **name** — Name of interface

character vector | string

Name of interface to be removed, specified as a character vector or string.

Example: "newInterface"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”

Term	Definition	Application	More Information
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2019a

### See Also

`addInterface` | `addValueType` | `addPhysicalInterface` | `addServiceInterface` | `getInterface` | `getInterfaceNames` | `Adapter`

### Topics

"Specify Physical Interfaces on Ports"

"Create Interfaces"

"Manage Interfaces with Data Dictionaries"

## removeProfile

**Package:** systemcomposer.arch

Remove profile from model

### Syntax

```
removeProfile(model,profileName)
```

### Description

removeProfile(model,profileName) removes the profile from a model.

### Examples

#### Remove Profile

Create a model.

```
model = systemcomposer.createModel("archModel",true);
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");  
latencybase = profile.addStereotype("LatencyBase");  
latencybase.addProperty("latency",Type="double");  
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");  
systemcomposer.profile.editor(profile)  
model.applyProfile("LatencyProfile");
```

Remove the profile from the model.

```
model.removeProfile("LatencyProfile");
```

### Input Arguments

#### **model** — Architecture model

model object

Architecture model, specified as a systemcomposer.arch.Model object.

#### **profileName** — Name of profile

character vector | string

Name of profile, specified as a character vector or string.

Example: "SystemProfile"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"



Term	Definition	Application	More Information
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

[applyProfile](#) | [createProfile](#)

### Topics

“Define Profiles and Stereotypes”

## removeProperty

**Package:** systemcomposer.profile

Remove property from stereotype

### Syntax

```
removeProperty(stereotype,propertyName)
```

### Description

removeProperty(stereotype,propertyName) removes a property from the stereotype.

### Examples

#### Remove Property

Add a component stereotype and add a VoltageRating property with value 5. Then remove the property.

```
profile = systemcomposer.profile.Profile.createProfile("myProfile");  
stereotype = addStereotype(profile,"electricalComponent",AppliesTo="Component")  
property = addProperty(stereotype,"VoltageRating",DefaultValue="5");  
removeProperty(stereotype,"VoltageRating")
```

### Input Arguments

#### stereotype — Stereotype

stereotype object

Stereotype, specified as a systemcomposer.profile.Stereotype object.

#### propertyName — Name of property

character vector | string

Name of property to be removed, specified as a character vector or string.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

addProperty | setProperty | getProperty

**Topics**

“Define Profiles and Stereotypes”

# removeQuery

**Package:** systemcomposer.view

Remove architecture view query

## Syntax

```
removeQuery(view, keepContents)
```

## Description

`removeQuery(view, keepContents)` removes the selection query and groupings on the view `view` with the option to keep contents (`keepContents`), which leaves the elements that were selected in the view. `removeQuery` allows for manually editing the view element by element. If `keepContents` is `true`, any property-based groupings are kept intact in the diagram but removed from `GroupBy`.

## Examples

### Remove Query From View and Keep Contents

Open the keyless entry system example and create a view. Specify the color as light blue, the query as all components, and group by the review status.

```
import systemcomposer.query.*

scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = model.createView("All Components Grouped by Review Status", ...
    Color="lightblue", Select=AnyComponent, ...
    GroupBy="AutoProfile.BaseComponent.ReviewStatus");
```

Open the Architecture Views Gallery to see the new view All Components Grouped by Review Status.

```
model.openViews
```

Remove the query and keep the contents. The view is now manually editable element by element, and the groupings are preserved.

```
view.removeQuery(true)
```

## Input Arguments

### **view** — Architecture view

view object

Architecture view, specified as a `systemcomposer.view.View` object.

**keepContents — Whether to keep contents in view**

true or 1 (default) | false or 0

Whether to keep contents in view, specified as a logical 1 (true) to keep contents specified by the removed selection query and property-based groupings or 0 (false) to remove all contents from the view.

**More About****Definitions**

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.  A viewpoint represents a stakeholder perspective that specifies the contents of the view.	“Modeling System Architecture of Keyless Entry System”
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• “Create Architecture Views Interactively”</li> <li>• “Create Architectural Views Programmatically”</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	“Find Elements in Model Using Queries”
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	“Inspect Components in Custom Architecture Views”

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

systemcomposer.view.View | createView | getView | deleteView | openViews | runQuery | modifyQuery | systemcomposer.view.ElementGroup

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

# removeReference

**Package:** `systemcomposer.interface`

Remove reference to dictionary

## Syntax

```
removeReference(dictionary, reference)
```

## Description

`removeReference(dictionary, reference)` removes a referenced dictionary from a dictionary in a System Composer model.

## Examples

### Remove Referenced Dictionary

Add a data interface `newInterface` to the local interface dictionary of the model. Save the local interface dictionary to a shared dictionary as an SLDD file.

```
arch = systemcomposer.createModel("newModel", true);  
addInterface(arch.InterfaceDictionary, "newInterface");  
saveToDictionary(arch, "TopDictionary")  
topDictionary = systemcomposer.openDictionary("TopDictionary.sldd");
```

Create a new dictionary and add it as a reference to the existing dictionary.

```
refDictionary = systemcomposer.createDictionary("ReferenceDictionary.sldd");  
addReference(topDictionary, "ReferenceDictionary.sldd")
```

Remove the referenced dictionary. Confirm in the **Model Explorer**.

```
removeReference(topDictionary, "ReferenceDictionary.sldd")
```

## Input Arguments

### **dictionary** — Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

### **reference** — Referenced dictionary

character vector | string

Referenced dictionary, specified as a character vector or string of the name of the referenced dictionary with the `.sldd` extension.



Example: "ReferenceDictionary.sldd"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• "Create Architecture Model with Interfaces and Requirement Links"</li> <li>• "Define Port Interfaces Between Components"</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• "Create Interfaces"</li> <li>• "Assign Interfaces to Ports"</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

**Version History**  
Introduced in R2021a

## **See Also**

saveToDictionary | createDictionary | openDictionary | linkDictionary |  
unlinkDictionary | addReference

## **Topics**

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## removeStereotype

**Package:** systemcomposer.profile

Remove stereotype from profile

### Syntax

```
removeStereotype(profile, stereotype)
```

### Description

removeStereotype(profile, stereotype) removes a stereotype from the specified profile.

### Examples

#### Remove Component Stereotype

Create a profile, add a component stereotype to the profile, open the **Profile Editor**, and remove the stereotype from the profile.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");  
stereotype = addStereotype(profile, "electricalComponent", AppliesTo="Component");  
systemcomposer.profile.editor  
profile.removeStereotype("electricalComponent")
```

### Input Arguments

#### profile — Profile

profile object

Profile, specified as a systemcomposer.profile.Profile object.

#### stereotype — Stereotype to remove

character vector | string | stereotype object

Stereotype to remove, specified as a systemcomposer.profile.Stereotype object or by name as a character vector or string.

Example: "electricalComponent"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

[getStereotype](#) | [addStereotype](#) | [getDefaultStereotype](#) | [setDefaultStereotype](#)

**Topics**

“Create a Profile and Add Stereotypes”

# removeStereotype

**Package:** systemcomposer.arch

Remove stereotype from model element

## Syntax

```
removeStereotype(element, stereotype)
```

## Description

`removeStereotype(element, stereotype)` removes a specified stereotype applied to a model element from the model element.

## Examples

### Remove Stereotype

Create a model with a component called Component.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
comp = addComponent(arch, "Component");
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

Apply the stereotype to the component, remove the stereotype from the component, and get the stereotypes on the component.

```
comp.applyStereotype("LatencyProfile.LatencyBase");
comp.removeStereotype("LatencyProfile.LatencyBase");
stereotypes = getStereotypes(comp)
```

```
stereotypes =
```

```
    1×0 empty cell array
```

## Input Arguments

### element — Architectural element

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### stereotype – Stereotype

character vector | string

Stereotype, specified as a character vector or string in the form "`<profile>.<stereotype>`". The profile must already be applied to the model.

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”



Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	"Extend Architectural Design Using Stereotypes"
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• "Set Properties"</li> <li>• "Add Properties with Stereotypes"</li> <li>• "Set Properties for Analysis"</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• "Define Profiles and Stereotypes"</li> <li>• "Use Stereotypes and Profiles"</li> </ul>

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	“Define Physical Ports on Component”

Term	Definition	Application	More Information
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2019a

## See Also

`applyStereotype` | `batchApplyStereotype` | `getStereotypes` | `getStereotypeProperties`

## Topics

"Remove Stereotypes"

## renameProfile

**Package:** systemcomposer.arch

Rename profile in model

### Syntax

```
renameProfile(model,oldProfileName,newProfileName)
```

### Description

`renameProfile(model,oldProfileName,newProfileName)` renames a profile on a model from `oldProfileName` to `newProfileName` to make it consistent if the name of the profile was changed in the file explorer.

---

**Note** Before you move, copy, or rename a profile to a different directory, you must close the profile in the **Profile Editor** or by using the `close` function. If you rename a profile, follow the example for the `renameProfile` function.

---

### Examples

#### Rename Profile

Create a model.

```
model = systemcomposer.createModel("archModel",true);
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

Save the model and close the model. Close the **Profile Editor**.

```
save(model)
close(model)
```

Save the profile.

```
save(profile)
```

Rename the profile in the file explorer to `LatencyProfileNew.xml`.

Load the model. Run the `renameProfile` API to update the model to refer to the correct renamed profile in the current directory.

```
model = systemcomposer.loadModel("archModel");
model.renameProfile("LatencyProfile", "LatencyProfileNew");
```

## Input Arguments

### **model** — Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

### **oldProfileName** — Old profile name

character vector | string

Old profile name, specified as a character vector or string.

Example: "MyProfile"

Data Types: char | string

### **newProfileName** — New profile name

character vector | string

New profile name, specified as a character vector or string.

Example: "MyProfileNew"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”



Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2020b

### See Also

[close](#) | [open](#) | [save](#)

### Topics

“Define Profiles and Stereotypes”

## resetParameterToDefault

**Package:** systemcomposer.arch

Reset parameter on component to default value

### Syntax

```
resetParameterToDefault(element,paramName)
```

### Description

`resetParameterToDefault(element,paramName)` resets parameter specified by `paramName` on the architectural element `element` to the default value and units, if applicable.

### Examples

#### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

**paramPressure.Type**

```
ans =
  ValueType with properties:
      Name: 'Pressure'
      DataType: 'double'
      Dimensions: '[1 1]'
      Units: 'psi'
      Complexity: 'real'
      Minimum: ''
      Maximum: ''
      Description: ''
      Owner: [1x1 systemcomposer.arch.Architecture]
      Model: [1x1 systemcomposer.arch.Model]
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'
      ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue =
'16'
```

```
paramUnits =
'in'
```

```
isDefault = logical
          1
```

```
paramName =
"Pressure"
```

```
paramValue =
'31'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
          0
```

```
paramName =
"Wear"
```

```
paramValue =
'0.25'
```

```
paramUnits =
'in'
```

```
isDefault = logical  
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue = 16
```

```

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 31

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Check the evaluated LeftWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

paramValue = 16

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 32

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```

[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

```

```
isDefault = logical
  1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'34'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =
  Parameter with properties:
    Name: 'Pressure'
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Component]
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '32'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce
noiseReduce =
  Parameter with properties:
    Name: "noiseReduction"
```

```
Value: '30'
Type: [1x1 systemcomposer.ValueType]
Parent: [1x1 systemcomposer.arch.Architecture]
Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the Muffler component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");
save(model)
save(topModel)
```

## Input Arguments

### **element** – Architectural element

architecture object | component object | variant component object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, or `systemcomposer.arch.VariantComponent` object.

### **paramName** – Parameter name

character vector | string

Parameter name, specified as a character vector or string.

Example: "GainArg"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>



Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	"Create Architecture Model with Interfaces and Requirement Links"
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022a

### See Also

[addParameter](#) | [getParameter](#) | [resetToDefault](#) | [getParameterPromotedFrom](#) | [getEvaluatedParameterValue](#) | [getParameterNames](#) | [setParameterValue](#) | [getParameterValue](#) | [setUnit](#)

### Topics

["Author Parameters in System Composer Using Parameter Editor"](#)  
["Access Model Arguments as Parameters on Reference Components"](#)  
["Use Parameters to Store Instance Values with Components"](#)

## resetToDefault

**Package:** systemcomposer.arch

Resets parameter value to default

### Syntax

```
resetToDefault(param)
```

### Description

`resetToDefault(param)` resets the parameter value on the instance `param` to its default value.

### Examples

#### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");  
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string  
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

```
paramPressure.Type
```

```

ans =
  ValueType with properties:
      Name: 'Pressure'
      DataType: 'double'
      Dimensions: '[1 1]'
      Units: 'psi'
      Complexity: 'real'
      Minimum: ''
      Maximum: ''
      Description: ''
      Owner: [1x1 systemcomposer.arch.Architecture]
      Model: [1x1 systemcomposer.arch.Model]
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'
      ExternalUID: ''

```

Get the RightWheel component parameter values.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

```

```

paramValue =
'16'

```

```

paramUnits =
'in'

```

```

isDefault = logical
           1

```

```

paramName =
"Pressure"

```

```

paramValue =
'31'

```

```

paramUnits =
'psi'

```

```

isDefault = logical
           0

```

```

paramName =
"Wear"

```

```

paramValue =
'0.25'

```

```

paramUnits =
'in'

```

```
isDefault = logical  
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue = 16
```

```

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 31

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Check the evaluated LeftWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

paramValue = 16

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 32

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```

[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

```

```
isDefault = logical  
  1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")  
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =  
'34'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
  0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
  1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";  
pressureParam
```

```
pressureParam =  
  Parameter with properties:  
  
    Name: "LeftWheel.Pressure"  
    Value: '30'  
    Type: [1x1 systemcomposer.ValueType]  
    Parent: [1x1 systemcomposer.arch.Architecture]  
    Unit: 'psi'
```



Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =
  Parameter with properties:
    Name: 'Pressure'
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Component]
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '32'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce
noiseReduce =
  Parameter with properties:
    Name: "noiseReduction"
```

```
Value: '30'
Type: [1x1 systemcomposer.ValueType]
Parent: [1x1 systemcomposer.arch.Architecture]
Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the `Muffler` component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");
save(model)
save(topModel)
```

## Input Arguments

### **param** — Parameter

parameter object

Parameter, specified as a `systemcomposer.arch.Parameter` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	<p>You can reuse compositions in the model using reference components. There are three types of reference components:</p> <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022b

### See Also

addParameter | getParameter | getParameterPromotedFrom |  
getEvaluatedParameterValue | getParameterNames | setParameterValue |  
resetParameterToDefault | getParameterValue | setUnit

### Topics

“Author Parameters in System Composer Using Parameter Editor”  
“Access Model Arguments as Parameters on Reference Components”  
“Use Parameters to Store Instance Values with Components”

## runQuery

**Package:** `systemcomposer.view`

Re-run architecture view query on model

### Syntax

```
runQuery(view)
```

### Description

`runQuery(view)` re-runs the existing query on the view `view`. This function removes elements that no longer match the query and adds elements that now match the query.

### Examples

#### Rerun Query on View

Open the keyless entry system example and create a view. Specify the color as light blue, and the query as all components.

```
import systemcomposer.query.*

scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
view = createView(model, "All Components", ...
    Color="lightblue", Select=AnyComponent);
```

Open the Architecture Views Gallery to see the new view All Components.

```
openViews(model)
```

Add components to the model. Rerun the query.

```
runQuery(view)
```

### Input Arguments

#### **view** — Architecture view

view object

Architecture view, specified as a `systemcomposer.view.View` object.

## More About

### Definitions

Term	Definition	Application	More Information
view	A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.	<p>You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.</p> <p>A viewpoint represents a stakeholder perspective that specifies the contents of the view.</p>	"Modeling System Architecture of Keyless Entry System"
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	<ul style="list-style-type: none"> <li>• "Create Architecture Views Interactively"</li> <li>• "Create Architectural Views Programmatically"</li> </ul>
query	A query is a specification that describes certain constraints or criteria to be satisfied by model elements.	Use queries to search elements with constraint criteria and to filter views.	"Find Elements in Model Using Queries"
component diagram	A component diagram represents a view with components, ports, and connectors based on how the model is structured.	Component diagrams allow you to programmatically or manually add and remove components from the view.	"Inspect Components in Custom Architecture Views"

Term	Definition	Application	More Information
hierarchy diagram	You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.	<p>There are two types of hierarchy diagrams:</p> <ul style="list-style-type: none"> <li>• <i>Component hierarchy diagrams</i> display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.</li> <li>• <i>Architecture hierarchy diagrams</i> display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.</li> </ul>	“Display Component Hierarchy and Architecture Hierarchy Using Views”

## Version History

Introduced in R2021a

### See Also

`systemcomposer.view.View` | `createView` | `getView` | `deleteView` | `openViews` | `removeQuery` | `modifyQuery` | `systemcomposer.view.ElementGroup` | `getQualifiedName`

### Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”



## save

**Package:** systemcomposer.profile

Save profile as file

### Syntax

```
filePath = save(profile)
filePath = save(profile,dirPath)
```

### Description

`filePath = save(profile)` saves a profile to disk as a file with a `.xml` extension to the current directory.

`filePath = save(profile,dirPath)` saves a profile to disk as a file with a `.xml` extension to the directory path `dirPath`.

### Examples

#### Save Profile

Create a profile named `newProfile` and save it in the current directory.

```
profile = systemcomposer.profile.Profile.createProfile("newProfile");
path = save(profile);
```

### Input Arguments

#### **profile** — Profile

profile object

Profile, specified as a `systemcomposer.profile.Profile` object.

#### **dirPath** — Path to save

character vector | string

Path to save, specified as a character vector or string. The current directory is the default if no path is specified.

Example: `"C:\Temp\MATLAB"`

Data Types: `char` | `string`

### Output Arguments

#### **filePath** — File path

character vector

File path where profile is saved, returned as a character vector.

## More About

### Definitions

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

## **See Also**

`createProfile` | `find` | `editor` | `open` | `load` | `close` | `closeAll`

## **Topics**

“Define Profiles and Stereotypes”

## save

**Package:** `systemcomposer.allocation`

Save allocation set as file

## Syntax

```
save(allocSet, dirPath)
```

## Description

`save(allocSet, dirPath)` saves the allocation set `allocSet` to disk as a file with an `.mldatx` extension. This function saves the file to the current directory if the optional input `dirPath` is left blank.

## Examples

### Create and Save Allocation Set

Create two new models with a component each.

```
mSource = systemcomposer.createModel("Source_Model_Allocation", true);  
sourceComp = addComponent(get(mSource, "Architecture"), "Source_Component");  
mTarget = systemcomposer.createModel("Target_Model_Allocation", true);  
targetComp = addComponent(get(mTarget, "Architecture"), "Target_Component");
```

Create the allocation set `MyNewAllocation`.

```
allocSet = systemcomposer.allocation.createAllocationSet("MyNewAllocation", ...  
    "Source_Model_Allocation", "Target_Model_Allocation");
```

Get the default allocation scenario.

```
defaultScenario = getScenario(allocSet, "Scenario 1");
```

Allocate components between models.

```
allocation = allocate(defaultScenario, sourceComp, targetComp);
```

Save the allocation set.

```
save(allocSet)
```

Open the **Allocation Editor**.

```
systemcomposer.allocation.editor
```

## Input Arguments

**allocSet** — Allocation set

allocation set object

Allocation set, specified as a `systemcomposer.allocation.AllocationSet` object.

**dirPath — Path to save**

character vector | string

Path to save, specified as a character vector or string. The current directory is the default if no path is specified.

Example: 'C:\Temp\MATLAB'

Data Types: char | string

**More About****Definitions**

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

**Version History**

Introduced in R2020b

**See Also**

createAllocationSet | createScenario | deleteScenario | getScenario | load | closeAll | close | find

**Topics**

“Create and Manage Allocations Programmatically”

## save

**Package:** `systemcomposer.arch`

Save architecture model or data dictionary

### Syntax

```
save(model)
save(dictionary)
```

### Description

`save(model)` saves the architecture model to a file specified in its `Name` property.

`save(dictionary)` saves the data dictionary.

### Examples

#### Save Model and Data Dictionary

```
arch = systemcomposer.createModel("newModel");
save(arch);
save(arch.InterfaceDictionary);
dictionary = systemcomposer.createDictionary("modelInterfaces.sldd");
dictionary.save;
```

### Input Arguments

#### **model** – Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

#### **dictionary** – Data dictionary

dictionary object

Data dictionary attached to the architecture model, specified as a `systemcomposer.interface.Dictionary` object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>



Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

`loadModel` | `close` | `systemcomposer.createModel`

### Topics

“Create Architecture Model”

“Manage Interfaces with Data Dictionaries”

## save

**Package:** systemcomposer.analysis

Save architecture instance

### Syntax

```
save(instance, fileName)
```

### Description

save(instance, fileName) saves an architecture instance to a MAT-file. The save method is part of the systemcomposer.analysis.ArchitectureInstance class.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The instance refers to the element instance on which the iteration is being performed.

---

### Examples

#### Save Architecture Instance to MAT-File

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Instantiate all stereotypes in a profile.

```
model = systemcomposer.createModel("archModel",true);
instance = instantiate(model.Architecture,"LatencyProfile","NewInstance");
```

Save the architecture instance.

```
instance.save("InstanceFile")
```

## Input Arguments

### **instance** — Architecture instance

architecture instance object

Architecture instance to be saved, specified as a `systemcomposer.analysis.ArchitectureInstance` object.

### **fileName** — MAT-file to save instance

character vector | string

MAT-file to save instance, specified as a character vector or string.

Example: "InstanceFile"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>

Term	Definition	Application	More Information
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	"Run Analysis Function"
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	"Create a Model Instance for Analysis"

## Version History

Introduced in R2019a

### See Also

instantiate | systemcomposer.analysis.Instance | loadInstance | deleteInstance | refresh | update | lookup | iterate

### Topics

"Write Analysis Function"

## saveAsModel

**Package:** `systemcomposer.arch`

(Not recommended) Save architecture of component to separate model

---

**Note** The `saveAsModel` function is not recommended. Use the `createArchitectureModel` function instead. For more information, see “Compatibility Considerations”.

---

### Syntax

```
saveAsModel(component, modelName)
```

### Description

`saveAsModel(component, modelName)` saves the architecture of the component to a separate architecture model and references the model from this component.

### Input Arguments

**component — Architecture component**

component object

Architecture component, specified as a `systemcomposer.arch.Component` object. The component must have an architecture with definition type `composition`. For other definition types, this function gives an error.

**modelName — Name of model**

character vector | string

Name of model, specified as a character vector or string.

Example: `"exMobileRobot"`

Data Types: `char` | `string`

### Version History

**Introduced in R2019a**

**R2021b\_plus: saveAsModel function is not recommended**

*Not recommended starting in R2021b\_plus*

The `saveAsModel` function is not recommended. Use the `createArchitectureModel` function instead.

### See Also

`linkToModel` | `isReference` | `createArchitectureModel` | `inlineComponent` | Reference Component

**Topics**

“Implement Component Behavior Using Simulink”  
“Decompose and Reuse Components”

## saveToDictionary

**Package:** `systemcomposer.arch`

Save interfaces to dictionary

### Syntax

```
saveToDictionary(model,dictionaryName)
saveToDictionary(dictionary,dictionaryName)
saveToDictionary( ____,Name,Value)
```

### Description

`saveToDictionary(model,dictionaryName)` saves all locally defined interfaces to a shared dictionary, and links the model to the shared dictionary with a `.sldd` extension.

`saveToDictionary(dictionary,dictionaryName)` saves all locally defined interfaces to a shared dictionary with an SLDD extension.

`saveToDictionary( ____,Name,Value)` saves all locally defined interfaces to a shared dictionary with additional options.

### Examples

#### Save to Dictionary

Create a model and a shared dictionary. Add an interface to the model's interface dictionary, and add an element. Save all interfaces defined in the model to the shared dictionary.

```
arch = systemcomposer.createModel("newModel",true);
dictionary = systemcomposer.createDictionary("myInterfaces.sldd");
interface = addInterface(arch.InterfaceDictionary,"newSignal");
element = addElement(interface,"newElement",Type="double");
saveToDictionary(arch,"myInterfaces")
```

### Input Arguments

#### **model** – Architecture model

model object

Architecture model, specified as a `systemcomposer.arch.Model` object.

#### **dictionary** – Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. You can specify the default data dictionary that defines local interfaces or an external data dictionary that carries interface definitions. If the model links to multiple data dictionaries, then `dictionary` must be the



dictionary that carries interface definitions. For information on how to create a dictionary, see `createDictionary`.

### **dictionaryName — Dictionary name**

character vector | string

Dictionary name, specified as a character vector or string. If a dictionary with this name does not exist, one will be created.

Example: "myInterfaces"

Data Types: char | string

### **Name-Value Pair Arguments**

Specify optional pairs of arguments as `Name1=Value1, . . . , NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

*Before R2021a, use commas to separate each name and value, and enclose Name in quotes.*

Example:

```
saveToDictionary(arch,"MyInterfaces",CollisionResolutionOption=systemcomposer
.interface.CollisionResolution.USE_MODEL)
```

### **CollisionResolutionOption — Option to resolve interface collisions using model or dictionary**

systemcomposer.interface.CollisionResolution.USE\_MODEL (default) |  
systemcomposer.interface.CollisionResolution.USE\_DICTIONARY

Option to resolve collisions using model or dictionary, specified as one of the following:

- `systemcomposer.interface.CollisionResolution.USE_MODEL` to prioritize interface duplicates using the local interfaces defined in the model.
- `systemcomposer.interface.CollisionResolution.USE_DICTIONARY` to prioritize interface duplicates using the interfaces defined in the saved dictionary.

Example:

```
saveToDictionary(arch,"MyInterfaces",CollisionResolutionOption=systemcomposer
.interface.CollisionResolution.USE_DICTIONARY)
```

Data Types: enum

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>

Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019b

### See Also

[createDictionary](#) | [linkDictionary](#) | [unlinkDictionary](#) | [openDictionary](#) | [addReference](#) | [removeReference](#)

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## setActiveChoice

**Package:** systemcomposer.arch

Set active choice on variant component

### Syntax

```
setActiveChoice(variantComponent, choice)
```

### Description

setActiveChoice(variantComponent, choice) sets the active choice on the variant component.

### Examples

#### Set Active Variant Choice

Create a model, get the root architecture, create one variant component, add two choices for the variant component, and set the active choice.

```
model = systemcomposer.createModel("archModel", true);  
arch = get(model, "Architecture");  
variant = addVariantComponent(arch, "Component1");  
compList = addChoice(variant, ["Choice1", "Choice2"]);  
setActiveChoice(variant, compList(2));
```

### Input Arguments

#### variantComponent — Variant component

variant component object

Variant component, specified as a systemcomposer.arch.VariantComponent object.

#### choice — Active choice in a variant component

component object | character vector | string

Active choice in a variant component, specified as a systemcomposer.arch.Component object or label of the variant choice as a character vector or string.

Example: "Choice2"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## Version History

Introduced in R2019a

### See Also

[addChoice](#) | [getActiveChoice](#) | [getChoices](#) | [addVariantComponent](#) | [Variant Component](#)

### Topics

"Create Variants"

## setAsynchronous

**Package:** `systemcomposer.interface`

Set function element as asynchronous

### Syntax

```
setAsynchronous(functionElem,isAsync)
```

### Description

`setAsynchronous(functionElem,isAsync)` sets the function element `functionElem` as asynchronous if `isAsync` is `true`.

### Examples

#### Set Function Element as Asynchronous

Create a new model.

```
model = systemcomposer.createModel("archModel","SoftwareArchitecture",true);
```

Create a service interface.

```
interface = addServiceInterface(model.InterfaceDictionary,"newServiceInterface");
```

Create a function element.

```
element = addElement(interface,"newFunctionElement");
```

Set function element as asynchronous.

```
setAsynchronous(element,true)
```

### Input Arguments

#### **functionElem** – Function element

function element object

Function element, specified as a `systemcomposer.interface.FunctionElement` object.

#### **isAsync** – Whether function element is asynchronous

`false` or `0` (default) | `true` or `1`

Whether function element is asynchronous, specified as a logical.

Data Types: `logical`



## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>“Author Service Interfaces for Client-Server Communication”</li> <li><code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2022b

### See Also

`addElement` | `createDictionary` | `addServiceInterface` | `getInterface` | `getInterfaceNames` | `removeInterface` | `linkDictionary` | `Adapter` | `addValueType` | `setFunctionPrototype` | `getFunctionArgument`

### Topics

“Author Service Interfaces for Client-Server Communication”

“Client-Server Interfaces in Class Diagram View”

“Define Port Interfaces Between Components”

# setComplexity

**Package:** systemcomposer

Set complexity for value type

## Syntax

```
setComplexity(valueType, complexity)
```

## Description

`setComplexity(valueType, complexity)` sets the complexity for the designated value type.

## Examples

### Set Complexity for Value Type

Create a model `archModel`.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName, true);
```

Add a value type `airSpeed` to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType("airSpeed");
```

Set the complexity for the value type as `complex`.

```
airSpeedType.setComplexity("complex")
```

## Input Arguments

### **valueType** — Value type, data element, or function argument

value type object | data element object | function argument object

Value type, data element, or function argument, specified as a `systemcomposer.ValueType`, `systemcomposer.interface.DataElement`, or `systemcomposer.interface.FunctionArgument` object.

### **complexity** — Complexity

"real" (default) | "complex" | "auto"

Complexity, specified as "real", "complex", or "auto".

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`createModel` | `addElement` | `addInterface` | `addValueType` | `createInterface` | `createOwnedType`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## setCondition

**Package:** systemcomposer.arch

Set condition on variant choice

### Syntax

```
setCondition(variantComponent,choice,expression)
```

### Description

`setCondition(variantComponent,choice,expression)` sets the variant control condition specified by `expression` for the choice `choice` on the variant component `variantComponent` to choose the active variant choice. If the condition is met on a variant choice, that variant choice becomes the active choice on the variant component.

### Examples

#### Set Variant Control Condition

Create a model, get the root architecture, create one variant component, add two choices for the variant component, and set a condition on one variant choice to choose the active variant choice.

```
model = systemcomposer.createModel("archModel",true);
arch = get(model,"Architecture");
mode = 1;
variant = addVariantComponent(arch,"Component1");
compList = addChoice(variant,["Choice1","Choice2"]);
setCondition(variant,compList(2),"mode == 2");
```

### Input Arguments

#### **variantComponent** – Variant component

variant component object

Variant component, specified as a `systemcomposer.arch.VariantComponent` object.

#### **choice** – Choice in variant component

component object

Choice in variant component, specified as a `systemcomposer.arch.Component` object.

#### **expression** – Control string

character vector | string

Control string that controls the selection of choice, specified as a character vector or string.

Data Types: `char` | `string`



## More About

### Definitions

Term	Definition	Application	More Information
variant	A variant is one of many structural or behavioral choices in a variant component.	Use variants to quickly swap different architectural designs for a component while performing analysis.	"Create Variants"
variant control	A variant control is a string that controls the active variant choice.	Set the variant control to programmatically control which variant is active.	"Set Variant Control Condition" on page 4-692

## Version History

Introduced in R2019a

### See Also

makeVariant | getCondition | addVariantComponent | addChoice | getActiveChoice | setActiveChoice | Variant Component

### Topics

"Create Variants"

## setDataType

**Package:** systemcomposer

Set data type for value type

### Syntax

```
setDataType(valueType, type)
```

### Description

`setDataType(valueType, type)` sets the data type for the designated value type.

### Examples

#### Set Data Type for Value Type

Create a model `archModel`.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName, true);
```

Add a value type `airSpeed` to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType("airSpeed");
```

Set the data type for the value type as `single`.

```
airSpeedType.setDataType("single")
```

### Input Arguments

#### **valueType** — Value type

value type object

Value type, specified as a `systemcomposer.ValueType` object.

#### **type** — Data type

character vector | string

Data type, specified as a character vector or string for a valid MATLAB data type.

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

# setDefaultComponentStereotype

**Package:** `systemcomposer.profile`

(Removed) Set default stereotype for components

---

**Note** The `setDefaultComponentStereotype` function has been removed. You can set a default component stereotype using the function `setDefaultElementStereotype`. For further details, see “Compatibility Considerations”.

---

## Syntax

```
setDefaultComponentStereotype(stereotype, stereotypeName)
```

## Description

`setDefaultComponentStereotype(stereotype, stereotypeName)` specifies the default stereotype `stereotypeName` of the child components whose parent component has `stereotype` applied.

## Input Arguments

### **stereotype — Stereotype**

stereotype object

Stereotype, specified as a `systemcomposer.profile.Stereotype` object.

### **stereotypeName — Default stereotype name**

character vector | string

Default stereotype name for child components, specified as a character vector or string in the form '`<profile>.<stereotype>`'.

Data Types: `char` | `string`

## Version History

**Introduced in R2019a**

### **R2021b: setDefaultComponentStereotype function has been removed**

*Errors starting in R2021b*

The `setDefaultComponentStereotype` function has been removed in R2021b. Use `setDefaultElementStereotype` instead.

## See Also

`applyStereotype` | `removeStereotype` | `setDefaultElementStereotype`

**Topics**

“Define Profiles and Stereotypes”

# setDefaultConnectorStereotype

**Package:** `systemcomposer.profile`

(Removed) Set default stereotype for connectors

---

**Note** The `setDefaultConnectorStereotype` function has been removed. You can set a default connector stereotype using the function `setDefaultElementStereotype`. For further details, see “Compatibility Considerations”.

---

## Syntax

```
setDefaultConnectorStereotype(stereotype, stereotypeName)
```

## Description

`setDefaultConnectorStereotype(stereotype, stereotypeName)` specifies the default stereotype `stereotypeName` of the connectors within a parent component that has `stereotype` applied.

## Input Arguments

### **stereotype — Stereotype**

stereotype object

Stereotype, specified as a `systemcomposer.profile.Stereotype` object.

### **stereotypeName — Default stereotype name**

character vector | string

Default stereotype name for connectors, specified as a character vector or string in the form '`<profile>.<stereotype>`'.

Data Types: `char` | `string`

## Version History

**Introduced in R2019a**

**R2021b: setDefaultConnectorStereotype function has been removed**

*Errors starting in R2021b*

The `setDefaultConnectorStereotype` function has been removed in R2021b. Use `setDefaultElementStereotype` instead.

## See Also

`applyStereotype` | `removeStereotype` | `setDefaultElementStereotype`

**Topics**

“Define Profiles and Stereotypes”



# setDefaultElementStereotype

**Package:** systemcomposer.profile

Set default stereotype for elements

## Syntax

```
setDefaultElementStereotype(stereotype, elementType, stereotypeName)
```

## Description

`setDefaultElementStereotype(stereotype, elementType, stereotypeName)` specifies the default stereotype `stereotypeName` of the child elements whose parent element of type `elementType` has the stereotype `stereotype` applied.

## Examples

### Set Default Component Stereotype

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency", ...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure", Type="boolean");
connLatency.addProperty("linkDistance", Type="double");

nodeLatency = profile.addStereotype("NodeLatency", ...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");

portLatency = profile.addStereotype("PortLatency", ...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth", Type="double");
portLatency.addProperty("dummy", Type="int32");

profile.save
```

Specify the `LatencyProfile.NodeLatency` stereotype as a component stereotype. Set the default component stereotype.

```
nodeLatency.AppliesTo = "Component";
nodeLatency.setDefaultElementStereotype("Component", "LatencyProfile.NodeLatency")
```

Create a model, apply the profile to the model, and add a parent component. Apply the parent component stereotype on the parent component. Then, open the **Profile Editor**.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName, true);
arch.applyProfile("LatencyProfile");
newComponent = addComponent(arch.Architecture, "Component");
```

```
newComponent.applyStereotype("LatencyProfile.NodeLatency");
systemcomposer.profile.editor(profile)
```

Create a child component and get the stereotypes on the child component.

```
childComponent = addComponent(newComponent.Architecture, "Child");
stereotypes = getStereotypes(childComponent)

stereotypes =

    1x1 cell array

    {'LatencyProfile.NodeLatency'}
```

### Set Default Port Stereotype

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency", ...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure", Type="boolean");
connLatency.addProperty("linkDistance", Type="double");

nodeLatency = profile.addStereotype("NodeLatency", ...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");

portLatency = profile.addStereotype("PortLatency", ...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth", Type="double");
portLatency.addProperty("dummy", Type="int32");

profile.save
```

Specify the `LatencyProfile.NodeLatency` stereotype as a component stereotype. Set the default port stereotype.

```
nodeLatency.AppliesTo = "Component";
nodeLatency.setDefaultElementStereotype("Port", "LatencyProfile.PortLatency");
```

Create a model, apply the profile to the model, and add a parent component. Apply the parent component stereotype on the parent component. Then, open the **Profile Editor**.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName, true);
arch.applyProfile("LatencyProfile");
newComponent = addComponent(arch.Architecture, "Component");
newComponent.applyStereotype("LatencyProfile.NodeLatency");
systemcomposer.profile.editor(profile)
```

Create an architecture port on the component and get the stereotypes on the port.

```
port = addPort(newComponent.Architecture, "testSig", "out");
stereotypes = getStereotypes(port)

stereotypes =

    1x1 cell array
```

```
{'LatencyProfile.PortLatency'}
```

## Set Default Connector Stereotype

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency",Type="double");
latencybase.addProperty("dataRate",Type="double",DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency",...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure",Type="boolean");
connLatency.addProperty("linkDistance",Type="double");

nodeLatency = profile.addStereotype("NodeLatency",...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources",Type="double",DefaultValue="1");

portLatency = profile.addStereotype("PortLatency",...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth",Type="double");
portLatency.addProperty("dummy",Type="int32");

profile.save
```

Specify the `LatencyProfile.NodeLatency` stereotype as a component stereotype. Set the default connector stereotype.

```
nodeLatency.AppliesTo = "Component";
nodeLatency.setDefaultElementStereotype("Connector","LatencyProfile.ConnectorLatency");
```

Create a model, apply the profile to the model, and add a parent component. Apply the parent component stereotype on the parent component. Then, open the **Profile Editor**.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName,true);
arch.applyProfile("LatencyProfile");
newComponent = addComponent(arch.Architecture,"Component");
newComponent.applyStereotype("LatencyProfile.NodeLatency");
systemcomposer.profile.editor(profile)
```

Create two child components. Add ports. Then, create a connection between the ports and get stereotypes on the connector.

```
childComponent1 = addComponent(newComponent.Architecture,"Child1");
childComponent2 = addComponent(newComponent.Architecture,"Child2");

outPort1 = addPort(childComponent1.Architecture,"testSig","out");
inPort1 = addPort(childComponent2.Architecture,"testSig","in");
srcPort = getPort(childComponent1,"testSig");
destPort = getPort(childComponent2,"testSig");

connector = connect(srcPort,destPort);
stereotypes = getStereotypes(connector)
```

```
stereotypes =
```

```
1x1 cell array
```

```
{'LatencyProfile.ConnectorLatency'}
```

## Input Arguments

### stereotype — Stereotype

stereotype object

Stereotype, specified as a `systemcomposer.profile.Stereotype` object.

### elementType — Element type

"Component" | "Port" | "Connector" | "Interface" | "Function"

Element type, specified as "Component", "Port", "Connector", "Interface", or "Function". The element type "Function" is only available for software architectures.

Data Types: char | string

### stereotypeName — Default stereotype name

character vector | string

Default stereotype name for child elements, specified as a character vector or string in the form "`<profile>.<stereotype>`".

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>

Term	Definition	Application	More Information
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2021b

### See Also

`applyStereotype` | `getDefaultElementStereotype` | `removeStereotype`

### Topics

“Define Profiles and Stereotypes”

# setDefaultPortStereotype

**Package:** `systemcomposer.profile`

(Removed) Set default stereotype for ports

---

**Note** The `setDefaultPortStereotype` function has been removed. You can set a default port stereotype using the function `setDefaultElementStereotype`. For further details, see “Compatibility Considerations”.

---

## Syntax

```
setDefaultPortStereotype(stereotype, stereotypeName)
```

## Description

`setDefaultPortStereotype(stereotype, stereotypeName)` specifies the default stereotype `stereotypeName` of the ports on the architecture of a parent component that has `stereotype` applied.

## Input Arguments

### **stereotype — Stereotype**

stereotype object

Stereotype, specified as a `systemcomposer.profile.Stereotype` object.

### **stereotypeName — Default stereotype name**

character vector | string

Default stereotype name for ports, specified as a character vector or string in the form '`<profile>.<stereotype>`'.

Data Types: `char` | `string`

## Version History

**Introduced in R2019a**

### **R2021b: setDefaultPortStereotype function has been removed**

*Errors starting in R2021b*

The `setDefaultPortStereotype` function has been removed in R2021b. Use `setDefaultElementStereotype` instead.

## See Also

`applyStereotype` | `removeStereotype` | `setDefaultElementStereotype`

**Topics**

“Define Profiles and Stereotypes”



# setDefaultStereotype

**Package:** systemcomposer.profile

Set default stereotype for profile

## Syntax

```
setDefaultStereotype(profile, name)
```

## Description

setDefaultStereotype(profile, name) sets the default stereotype with name name for a profile profile. The stereotype must apply to components.

## Examples

### Set Default Stereotype

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency", ...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure", Type="boolean");
connLatency.addProperty("linkDistance", Type="double");

nodeLatency = profile.addStereotype("NodeLatency", ...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");

portLatency = profile.addStereotype("PortLatency", ...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth", Type="double");
portLatency.addProperty("dummy", Type="int32");

profile.save
```

Set the default stereotype.

```
profile.setDefaultStereotype("NodeLatency")
```

Create a model and apply the profile to the model. Open the **Profile Editor**.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName, true);
arch.applyProfile(LatencyProfile);
systemcomposer.profile.editor
```

Get stereotypes on the root architecture.

```
stereotypes = getStereotypes(arch.Architecture)

stereotypes =
```

1x1 cell array

```
{'LatencyProfile.NodeLatency'}
```

## Input Arguments

### profile — Profile

profile object

Profile, specified as a `systemcomposer.profile.Profile` object.

### name — Stereotype name

character vector | string

Stereotype name, specified as a character vector or string. The name of the stereotype must be unique within the profile.

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>“Compose Architectures Visually”</li> <li>“Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>Extract the root-level architecture contained in the model.</li> <li>Apply profiles.</li> <li>Link interface data dictionaries.</li> <li>Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”

Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

createProfile | getDefaultStereotype | addStereotype | getStereotype | removeStereotype

**Topics**

“Create a Profile and Add Stereotypes”

## setDescription

**Package:** systemcomposer

Set description for value type or interface

### Syntax

```
setDescription(valueType,description)
setDescription(interface,description)
```

### Description

setDescription(valueType,description) sets the description for the designated value type.

setDescription(interface,description) sets the description for the designated interface.

### Examples

#### Set Description for Value Type

Create a model archModel.

```
modelName = "archModel";
arch = systemcomposer.createModel(modelName,true);
```

Add a value type airSpeed to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType("airSpeed");
```

Set the description for the value type as Maintain altitude.

```
airSpeedType.setDescription("Maintain altitude")
```

### Input Arguments

#### valueType — Value type, data element, or function argument

value type object | data element object | function argument object

Value type, data element, or function argument, specified as a systemcomposer.ValueType, systemcomposer.interface.DataElement, or systemcomposer.interface.FunctionArgument object.

#### interface — Interface

data interface object | physical interface object | service interface object

Interface, specified as a systemcomposer.interface.DataInterface, systemcomposer.interface.PhysicalInterface, or systemcomposer.interface.ServiceInterface object.

#### description — Description

character vector | string

Description, specified as a character vector or string.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b



**See Also**

createModel | addValueType | addElement | addInterface | createInterface | createOwnedType

**Topics**

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## setDimensions

**Package:** systemcomposer

Set dimensions for value type

### Syntax

```
setDimensions(valueType,dimensions)
```

### Description

`setDimensions(valueType,dimensions)` sets the dimensions for the designated value type.

### Examples

#### Set Dimensions for Value Type

Create a model `archModel`.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName,true);
```

Add a value type `airSpeed` to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType("airSpeed");
```

Set the dimensions for the value type as 2.

```
airSpeedType.setDimensions("2")
```

### Input Arguments

#### **valueType** — Value type, data element, or function argument

value type object | data element object | function argument object

Value type, data element, or function argument, specified as a `systemcomposer.ValueType`, `systemcomposer.interface.DataElement`, or `systemcomposer.interface.FunctionArgument` object.

#### **dimensions** — Dimensions

character vector | string

Dimensions, specified as a character vector or string.

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

# setFunctionPrototype

**Package:** `systemcomposer.interface`

Set prototype for function element

## Syntax

```
setFunctionPrototype(functionElem,prototype)
```

## Description

`setFunctionPrototype(functionElem,prototype)` sets the prototype for a function represented by the function element object `functionElem`. Use prototypes to add, remove, and rename the arguments of a function element.

## Examples

### Set Function Prototype

Create a new model.

```
model = systemcomposer.createModel("archModel","SoftwareArchitecture",true)
```

Create a service interface.

```
interface = addServiceInterface(model.InterfaceDictionary,"newServiceInterface")
```

Create a function element.

```
element = addElement(interface,"f0")
```

Set the function prototype.

```
setFunctionPrototype(element,"y=f0(u)")
```

## Input Arguments

### **functionElem** – Function element

function element object

Function element, specified as a `systemcomposer.interface.FunctionElement` object.

### **prototype** – Prototype

character vector | string

Prototype, specified as a character vector or string in the form `[y1,y2]=f0(u1,u2)` where `y1` and `y2` are output arguments, `u1` and `u2` are input arguments, and `f0` is the name of the `functionElem` object.

Example: `"y=f0(u1,u2)"`

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.	Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.	<ul style="list-style-type: none"> <li>“Author Software Architectures”</li> <li>“Simulate and Deploy Software Architectures”</li> </ul>
software component	A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.	Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Create Software Architecture from Component”</li> </ul>
software composition	A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.	Encapsulate functionality by aggregating or nesting multiple software components or compositions.	“Modeling Software Architecture of Throttle Position Control System”
function	A function is an entry point that can be defined in a software component.	You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the <b>Functions Editor</b> .	“Author and Extend Functions for Software Architectures”
service interface	A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.	Once you have defined a service interface in the <b>Interface Editor</b> , you can assign it to client and server ports using the <b>Property Inspector</b> . You can also use the <b>Property Inspector</b> to assign stereotypes to service interfaces.	<ul style="list-style-type: none"> <li>“Author Service Interfaces for Client-Server Communication”</li> <li><code>systemcomposer.interface.ServiceInterface</code></li> </ul>

Term	Definition	Application	More Information
function element	A function element describes the attributes of a function in a client-server interface.	<p>Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:</p> <ul style="list-style-type: none"> <li>• Synchronous execution <ul style="list-style-type: none"> <li>– When the client calls the server, the function runs immediately and returns the output arguments to the client.</li> </ul> </li> <li>• Asynchronous execution <ul style="list-style-type: none"> <li>– When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the <b>Functions Editor</b> and <b>Schedule Editor</b> and returns the output arguments to the client.</li> </ul> </li> </ul>	systemcomposer.interface.FunctionElement
function argument	A function argument describes the attributes of an input or output argument in a function element.	You can set the properties of a function argument in the <b>Interface Editor</b> just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.	systemcomposer.interface.FunctionArgument
class diagram	A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.	Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.	“Class Diagram View of Software Architectures”

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”



Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2022a

### See Also

`addElement` | `createDictionary` | `addServiceInterface` | `getInterface` | `getInterfaceNames` | `removeInterface` | `linkDictionary` | `Adapter` | `addValueType` | `getFunctionArgument` | `setAsynchronous`

### Topics

“Author Service Interfaces for Client-Server Communication”

“Client-Server Interfaces in Class Diagram View”

“Define Port Interfaces Between Components”

## setInterface

**Package:** `systemcomposer.arch`

Set interface for port

### Syntax

```
setInterface(port, interface)
```

### Description

`setInterface(port, interface)` sets the interface for a port.

### Examples

#### Set Interface for Port and Remove Interface on Port

Create a model and get the root architecture.

```
model = systemcomposer.createModel("archModel", true);
rootArch = get(model, "Architecture");
```

Add a component and add a port to the component.

```
newComponent = addComponent(rootArch, "newComponent");
newPort = addPort(newComponent.Architecture, "newPort", "in");
```

Add a data interface and set the interface for the port.

```
newInterface = addInterface(model.InterfaceDictionary, "newInterface");
setInterface(newPort, newInterface)
```

Remove the data interface on the port.

```
newPort.setInterface("")
```

### Input Arguments

#### port — Port

port object

Port, specified as a `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` object.

#### interface — Interface

data interface object | value type object | physical interface object | service interface object | empty string | empty character vector

Interface to set, specified as a `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object. Passing in an empty string or character vector removes the interface on the port.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• "Create Architecture Model with Interfaces and Requirement Links"</li> <li>• "Define Port Interfaces Between Components"</li> </ul>

Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `addPhysicalInterface` | `addServiceInterface`

### Topics

- “Specify Physical Interfaces on Ports”
- “Create Interfaces”
- “Manage Interfaces with Data Dictionaries”

# setMaximum

**Package:** systemcomposer

Set maximum for value type

## Syntax

```
setMaximum(valueType,maximum)
```

## Description

setMaximum(valueType,maximum) sets the maximum for the designated value type.

## Examples

### Set Maximum for Value Type

Create a model archModel.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName,true);
```

Add a value type airSpeed to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType("airSpeed");
```

Set the maximum for the value type as 100.

```
airSpeedType.setMaximum("100")
```

## Input Arguments

### valueType — Value type, data element, or function argument

value type object | data element object | function argument object

Value type, data element, or function argument, specified as a `systemcomposer.ValueType`, `systemcomposer.interface.DataElement`, or `systemcomposer.interface.FunctionArgument` object.

### maximum — Maximum

character vector | string

Maximum, specified as a character vector or string.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”



Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## setMinimum

**Package:** systemcomposer

Set minimum for value type

### Syntax

```
setMinimum(valueType,minimum)
```

### Description

setMinimum(valueType,minimum) sets the minimum for the designated value type.

### Examples

#### Set Minimum for Value Type

Create a model archModel.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName,true);
```

Add a value type airSpeed to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType("airSpeed");
```

Set the minimum for the value type as 0.

```
airSpeedType.setMinimum("0")
```

### Input Arguments

#### valueType — Value type, data element, or function argument

value type object | data element object | function argument object

Value type, data element, or function argument, specified as a `systemcomposer.ValueType`, `systemcomposer.interface.DataElement`, or `systemcomposer.interface.FunctionArgument` object.

#### minimum — Minimum

character vector | string

Minimum, specified as a character vector or string.

Data Types: `char` | `string`

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

# setName

**Package:** systemcomposer.arch

Set name for port

## Syntax

```
setName(port, name)
```

## Description

setName(port, name) sets the name for the designated port.

## Examples

### Set New Name for Port

Create a model, get the root architecture, add a component, add a port, and set a new name for the port.

```
model = systemcomposer.createModel("archModel", true);
rootArch = get(model, "Architecture");
newComponent = addComponent(rootArch, "newComponent");
newPort = addPort(newComponent.Architecture, "newCompPort", "in");
setName(newPort, "compPort")
```

## Input Arguments

### port — Port

port object

Port, specified as a systemcomposer.arch.ArchitecturePort or systemcomposer.arch.ComponentPort object.

### name — Name of port

character vector | string

Name of port, specified as a character vector or string.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

## Version History

Introduced in R2019a

### See Also

Component | `systemcomposer.arch.ArchitecturePort` |  
`systemcomposer.arch.ComponentPort`

## setName

**Package:** systemcomposer.interface

Set name for value type, function argument, interface, or element

### Syntax

```
setName(interfaceElem, name)
```

### Description

`setName(interfaceElem, name)` sets the name for the designated value type, interface, element, or function argument.

### Examples

#### Set Name for Data Element

Create a model `archModel`.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName, true);
```

Add a data interface, then create an data element `x`.

```
interface = arch.InterfaceDictionary.addInterface("interface");  
elem = interface.addElement("x");
```

Set a new name for the data element as `newName`.

```
setName(elem, "newName");
```

### Input Arguments

#### **interfaceElem** — Value type, function argument, interface, or element

data interface object | data element object | physical interface object | physical element object | value type object | service interface object | function element object | function argument object

Value type, function argument, interface, or element to be named, specified as a `systemcomposer.interface.DataInterface`, `systemcomposer.interface.DataElement`, `systemcomposer.interface.PhysicalInterface`, `systemcomposer.interface.PhysicalElement`, `systemcomposer.ValueType`, `systemcomposer.interface.ServiceInterface`, `systemcomposer.interface.FunctionElement`, or `systemcomposer.interface.FunctionArgument` object.

#### **name** — Name

character vector | string

Name of value type, function argument, interface, or element, specified as a character vector or string. This name must be a valid MATLAB identifier.



Example: "newName"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>

Term	Definition	Application	More Information
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

**See Also**

createModel | addElement | addInterface | addPhysicalInterface | addValueType | addServiceInterface

**Topics**

“Specify Physical Interfaces on Ports”

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## setParameterValue

**Package:** systemcomposer.arch

Set value of parameter

### Syntax

```
setParameterValue(element, paramName, value, unit)
```

### Description

`setParameterValue(element, paramName, value, unit)` sets the parameter value specified by `value` and, optionally, the parameter units `unit` for a specified parameter name, `paramName`, on an architectural element, `element`.

### Examples

#### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model, Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model, Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

**paramPressure.Type**

```
ans =
  ValueType with properties:
      Name: 'Pressure'
      DataType: 'double'
      Dimensions: '[1 1]'
      Units: 'psi'
      Complexity: 'real'
      Minimum: ''
      Maximum: ''
      Description: ''
      Owner: [1x1 systemcomposer.arch.Architecture]
      Model: [1x1 systemcomposer.arch.Model]
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'
      ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue =
'16'
```

```
paramUnits =
'in'
```

```
isDefault = logical
    1
```

```
paramName =
"Pressure"
```

```
paramValue =
'31'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
    0
```

```
paramName =
"Wear"
```

```
paramValue =
'0.25'
```

```
paramUnits =
'in'
```

```
isDefault = logical  
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'32'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue = 16
```

```

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 31

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Check the evaluated LeftWheel parameters.

```

for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))
end

```

```

paramName =
"Diameter"

paramValue = 16

paramUnits =
'in'

paramName =
"Pressure"

paramValue = 32

paramUnits =
'psi'

paramName =
"Wear"

paramValue = 0.2500

paramUnits =
'in'

```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```

[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")

paramValue =
'32'

paramUnits =
'psi'

```

```
isDefault = logical
  1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'34'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
  1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```



Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =
  Parameter with properties:
    Name: 'Pressure'
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Component]
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);
pressureParam
```

```
pressureParam =
  Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '32'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce
noiseReduce =
  Parameter with properties:
    Name: "noiseReduction"
```

```
Value: '30'  
Type: [1x1 systemcomposer.ValueType]  
Parent: [1x1 systemcomposer.arch.Architecture]  
Unit: 'dB'
```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the Muffler component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");  
save(model)  
save(topModel)
```

## Input Arguments

### **element** — Architectural element

architecture object | component object | variant component object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, or `systemcomposer.arch.VariantComponent` object.

### **paramName** — Parameter name

character vector | string

Parameter name, specified as a character vector or string.

Example: "GainArg"

Data Types: char | string

### **value** — Parameter value

character vector | string

Parameter value, specified as a character vector or string.

Data Types: char | string

### **unit** — Units of parameter

character vector | string

Units of parameter, specified as a character vector or string. You can change the units of a parameter only if the value type specifies a unit.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• "Implement Component Behavior Using Simulink"</li> <li>• "Create Architecture Reference"</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• "Author Parameters in System Composer Using Parameter Editor"</li> <li>• "Access Model Arguments as Parameters on Reference Components"</li> <li>• "Use Parameters to Store Instance Values with Components"</li> </ul>

Term	Definition	Application	More Information
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>“Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>“Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>“Implement Behaviors for Architecture Model Simulation”</li> <li>“Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022a

### See Also

addParameter | getParameter | resetToDefault | getParameterPromotedFrom | getEvaluatedParameterValue | getParameterNames | getParameterValue | setUnit | resetParameterToDefault

### Topics

“Author Parameters in System Composer Using Parameter Editor”

“Access Model Arguments as Parameters on Reference Components”

“Use Parameters to Store Instance Values with Components”

## setProperty

**Package:** systemcomposer.arch

Set property value corresponding to stereotype applied to element

### Syntax

```
setProperty(element, propertyName, propertyValue, propertyUnits)
```

### Description

setProperty(element, propertyName, propertyValue, propertyUnits) sets the value and units of the property specified in the propertyName argument. Set the property corresponding to an applied stereotype by qualified name "<profile>.<stereotype>.<property>".

### Examples

#### Apply a Stereotype and Set Numeric Property Value

In this example, weight is a property of the stereotype sysComponent.

Create a model with a component called "Component".

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
comp = addComponent(arch, "Component");
```

Create a profile with a stereotype and properties, open the **Profile Editor**, then apply the profile to the model.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");
latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");
systemcomposer.profile.editor(profile)
model.applyProfile("LatencyProfile");
```

Apply the stereotype to the component, and set a new latency property.

```
applyStereotype(comp, "LatencyProfile.LatencyBase")
setProperty(comp, "LatencyProfile.LatencyBase.latency", "500")
```

#### Apply a Stereotype and Set String Property Value

In this example, description is a property of the stereotype sysComponent.

Create a model with a component called Component.

```
model = systemcomposer.createModel("archModel", true);
arch = get(model, "Architecture");
comp = addComponent(arch, "Component");
```

Create a profile with a stereotype, then apply the profile to the model. Open the **Profile Editor**.

```
profile = systemcomposer.profile.Profile.createProfile("sysProfile");
base = profile.addStereotype("sysComponent");
base.addProperty("description",Type="string");
model.applyProfile("sysProfile");
systemcomposer.profile.editor
```

Apply the stereotype to the component, and set a new description property.

```
applyStereotype(comp,"sysProfile.sysComponent")
expression = sprintf("%s',"component description")
setProperty(comp,"sysProfile.sysComponent.description",expression)
```

## Set Property Value on Existing Component

Set the `AutoProfile.System.Cost` property on the `FOB Locator System` component.

Launch the keyless entry system project.

```
scKeyLessEntrySystem
```

Load the model and find the `FOB Locator System` component.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
comp = lookup(model,Path="KeylessEntryArchitecture/FOB Locator System");
```

Set the `Cost` property on the component.

```
setProperty(comp,"AutoProfile.System.Cost","200","USD")
```

## Input Arguments

### element — Architectural element

architecture object | component object | port object | connector object | physical connector object | function object | data interface object | value type object | physical interface object | service interface object

Architectural element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.arch.Function`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, `systemcomposer.interface.PhysicalInterface`, or `systemcomposer.interface.ServiceInterface` object.

### propertyName — Name of property

character vector | string

Name of property, specified as a character vector or string in the form '`<profile>.<stereotype>.<property>`'.

Data Types: `char` | `string`

### propertyValue — Value of property

character vector | string

Value of property, specified as a character vector or string. Specify string values in the form `sprintf('%s', '<contents of string>')`. For more information, see “Apply a Stereotype and Set String Property Value” on page 4-754.

Data Types: `char` | `string`

### **propertyUnits – Units of property**

character vector | string

Units of property to interpret property values, specified as a character vector or string.

Data Types: `char` | `string`

## **More About**

### **Definitions**

<b>Term</b>	<b>Definition</b>	<b>Application</b>	<b>More Information</b>
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”



Term	Definition	Application	More Information
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	<p>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:</p> <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	"Components"
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	<p>There are different types of ports:</p> <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

Term	Definition	Application	More Information
physical subsystem	A physical subsystem is a Simulink subsystem with Simscape connections.	A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.	“Implement Component Behavior Using Simscape”

Term	Definition	Application	More Information
physical port	A physical port represents a Simscape physical modeling connector port called a Connection Port.	Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.	"Define Physical Ports on Component"
physical connector	A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.	Use physical connectors to connect physical components that represent features of a system to simulate mathematically.	"Architecture Model with Simscape Behavior for a DC Motor"
physical interface	A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a <code>Simulink.ConnectionBus</code> object that specifies any number of <code>Simulink.ConnectionElement</code> objects.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on Ports"
physical element	A physical element describes the decomposition of a physical interface. A physical element is equivalent to a <code>Simulink.ConnectionElement</code> object.	Define the Type of a physical element as a physical domain to enable use of that domain in a physical model.	"Describe Component Behavior Using Simscape"

## Version History

Introduced in R2019a

### See Also

getProperty | addProperty | removeProperty

### Topics

"Set Properties for Analysis"

## setType

**Package:** systemcomposer.interface

Set shared type on data element or function argument

### Syntax

```
setType(dataElement, type)
```

### Description

setType(dataElement, type) sets a type on a data element or a function argument.

### Examples

#### Set Value Type on Data Element

```
model = systemcomposer.createModel("archModel", true);  
dictionary = model.InterfaceDictionary;  
airspeedType = dictionary.addValueType("AirSpeed");  
port = model.Architecture.addPort("inPort", "in");  
interface = port.createInterface("DataInterface");  
element = interface.addElement("newElement");  
element.setType(airspeedType)
```

Open the **Interface Editor** from the **Modeling > Design** menu. Observe the new value type `AirSpeed` under the model `archModel.slx` interface dictionary. Switch from **Dictionary View** to **Port Interface View** on the right. Observe the owned data element on the port interface `inPort` called `newElement` with Type defined as `AirSpeed`.

### Input Arguments

#### **dataElement** – Data element or function argument

data element object | function argument object

Data element, specified as a `systemcomposer.interface.DataElement` or `systemcomposer.interface.FunctionArgument` object.

#### **type** – Type

data interface object | value type object

Type, specified as a `systemcomposer.interface.DataInterface`, for data elements only, or `systemcomposer.ValueType` object.

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`addValueType` | `createModel` | `addInterface` | `createOwnedType` | `createInterface` | `removeInterface`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

# setUnit

**Package:** systemcomposer.arch

Set units on parameter value

## Syntax

```
setUnit(arch,paramName,unit)
```

## Description

`setUnit(arch,paramName,unit)` sets the units specified by `unit` for the parameter specified by `paramName` for the architectural element `arch`. You cannot set units for a parameter promoted from a component.

## Examples

### Modify Parameters for Axle Architecture

This example shows a wheel axle architecture model with instance-specific parameters exposed in System Composer™. These parameters are defined as model arguments on the Simulink® reference model used as a model behavior linked to two System Composer components. You can change the values of these parameters independently on each reference component.

To add parameters to the architecture model or components, use the Parameter Editor. To remove these parameters, delete them from the **Parameter Editor**.

Open the architecture model of the wheel axle `mAxleArch` to interact with the parameters on the reference components using the Property Inspector.

```
model = systemcomposer.openModel("mAxleArch");
```

Look up the Component objects for the `RightWheel` and `LeftWheel` components.

```
rightWheelComp = lookup(model,Path="mAxleArch/RightWheel");
leftWheelComp = lookup(model,Path="mAxleArch/LeftWheel");
```

Get the parameter names for the `RightWheel` component. Since the `LeftWheel` component is linked to the same reference model `mWheel`, the parameters are the same on the `LeftWheel` component.

```
paramNames = rightWheelComp.getParameterNames
```

```
paramNames = 1x3 string
    "Diameter"    "Pressure"    "Wear"
```

Get the `Pressure` parameter on the `RightWheel` component architecture.

```
paramPressure = rightWheelComp.Architecture.getParameter(paramNames(2));
```

Display the value type for the `Pressure` parameter.

paramPressure.Type

```
ans =  
  ValueType with properties:  
  
      Name: 'Pressure'  
      DataType: 'double'  
      Dimensions: '[1 1]'  
      Units: 'psi'  
      Complexity: 'real'  
      Minimum: ''  
      Maximum: ''  
      Description: ''  
      Owner: [1x1 systemcomposer.arch.Architecture]  
      Model: [1x1 systemcomposer.arch.Model]  
      UUID: '47c2446a-f6b0-4710-9a73-7ed25d1671c4'  
      ExternalUID: ''
```

Get the RightWheel component parameter values.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits,isDefault] = rightWheelComp.getParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"
```

```
paramValue =  
'16'
```

```
paramUnits =  
'in'
```

```
isDefault = logical  
    1
```

```
paramName =  
"Pressure"
```

```
paramValue =  
'31'
```

```
paramUnits =  
'psi'
```

```
isDefault = logical  
    0
```

```
paramName =  
"Wear"
```

```
paramValue =  
'0.25'
```

```
paramUnits =  
'in'
```



```
isDefault = logical
    1
```

Get the LeftWheel component parameter values.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue =
'16'
```

```
paramUnits =
'in'
```

```
isDefault = logical
    1
```

```
paramName =
"Pressure"
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
    1
```

```
paramName =
"Wear"
```

```
paramValue =
'0.25'
```

```
paramUnits =
'in'
```

```
isDefault = logical
    1
```

First, check the evaluated RightWheel parameters.

```
for i = 1:length(paramNames)
    paramName = paramNames(i)
    [paramValue,paramUnits] = rightWheelComp.getEvaluatedParameterValue(paramNames(i))
end
```

```
paramName =
"Diameter"
```

```
paramValue = 16
```

```
paramUnits =  
'in'  
  
paramName =  
"Pressure"  
  
paramValue = 31  
  
paramUnits =  
'psi'  
  
paramName =  
"Wear"  
  
paramValue = 0.2500  
  
paramUnits =  
'in'
```

Check the evaluated LeftWheel parameters.

```
for i = 1:length(paramNames)  
    paramName = paramNames(i)  
    [paramValue,paramUnits] = leftWheelComp.getEvaluatedParameterValue(paramNames(i))  
end
```

```
paramName =  
"Diameter"  
  
paramValue = 16  
  
paramUnits =  
'in'  
  
paramName =  
"Pressure"  
  
paramValue = 32  
  
paramUnits =  
'psi'  
  
paramName =  
"Wear"  
  
paramValue = 0.2500  
  
paramUnits =  
'in'
```

Set the parameter value and unit for the PSI parameter on the LeftWheel component.

First, check the current values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")  
  
paramValue =  
'32'  
  
paramUnits =  
'psi'
```

```
isDefault = logical
1
```

Update the values for the pressure on LeftWheel.

```
leftWheelComp.setParameterValue("Pressure", "34")
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'34'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
0
```

Revert the Pressure parameter on LeftWheel to its default value.

```
leftWheelComp.resetParameterToDefault("Pressure")
```

Check the reverted values for the pressure on LeftWheel.

```
[paramValue,paramUnits,isDefault] = leftWheelComp.getParameterValue("Pressure")
```

```
paramValue =
'32'
```

```
paramUnits =
'psi'
```

```
isDefault = logical
1
```

Promote the Pressure parameter on the LeftWheel component.

```
addParameter(model.Architecture,Path="mAxleArch/LeftWheel",Parameters="Pressure");
```

Get the promoted Pressure parameter from the root architecture of the mAxleArch model.

```
pressureParam = model.Architecture.getParameter("LeftWheel.Pressure");
```

Adjust the value of the promoted Pressure parameter.

```
pressureParam.Value = "30";
pressureParam
```

```
pressureParam =
Parameter with properties:
    Name: "LeftWheel.Pressure"
    Value: '30'
    Type: [1x1 systemcomposer.ValueType]
    Parent: [1x1 systemcomposer.arch.Architecture]
    Unit: 'psi'
```

Get the source parameter from which the Pressure parameter is promoted.

```
sourceParam = getParameterPromotedFrom(pressureParam)
```

```
sourceParam =  
  Parameter with properties:  
  
    Name: 'Pressure'  
    Value: '30'  
    Type: [1x1 systemcomposer.ValueType]  
    Parent: [1x1 systemcomposer.arch.Component]  
    Unit: 'psi'
```

Reset the value of the promoted Pressure parameter to the default value in the source parameter.

```
resetToDefault(pressureParam);  
pressureParam
```

```
pressureParam =  
  Parameter with properties:  
  
    Name: "LeftWheel.Pressure"  
    Value: '32'  
    Type: [1x1 systemcomposer.ValueType]  
    Parent: [1x1 systemcomposer.arch.Architecture]  
    Unit: 'psi'
```

Delete the promoted parameter.

```
destroy(pressureParam)
```

Add a new Muffler component to the mAxleArch architecture model.

```
topModel = systemcomposer.loadModel("mAxleArch");  
mufflerComp = addComponent(topModel.Architecture, "Muffler");
```

Add the parameter noiseReduction to the Muffler component.

```
noiseReduce = addParameter(mufflerComp.Architecture, "noiseReduction");
```

Set the default Unit value for the NoiseReduction parameter.

```
valueTypeNoise = noiseReduce.Type;  
valueTypeNoise.Units = "dB";
```

Set the Value property for the noiseReduction parameter.

```
noiseReduce.Value = "30";
```

View the properties of the noiseReduction parameter.

```
noiseReduce  
  
noiseReduce =  
  Parameter with properties:  
  
    Name: "noiseReduction"
```

```

Value: '30'
Type: [1x1 systemcomposer.ValueType]
Parent: [1x1 systemcomposer.arch.Architecture]
Unit: 'dB'

```

Rearrange the `mAxleArch` architecture model to view all components.

```
Simulink.BlockDiagram.arrangeSystem("mAxleArch");
```

Delete the `Muffler` component.

```
destroy(mufflerComp)
```

Save the updated models.

```
model = systemcomposer.loadModel("mWheelArch");
save(model)
save(topModel)
```

## Input Arguments

### **arch** — Architecture

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object.

### **paramName** — Parameter name

character vector | string

Parameter name, specified as a character vector or string.

Example: "GainArg"

Data Types: char | string

### **unit** — Units of parameter

character vector | string

Units of parameter, specified as a character vector or string. You can change the units of a parameter only if the value type specifies a unit.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	“Ports”
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	“Connections”

Term	Definition	Application	More Information
reference component	A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.	You can reuse compositions in the model using reference components. There are three types of reference components: <ul style="list-style-type: none"> <li>• <i>Model references</i> are Simulink models.</li> <li>• <i>Subsystem references</i> are Simulink subsystems.</li> <li>• <i>Architecture references</i> are System Composer architecture models or subsystems.</li> </ul>	<ul style="list-style-type: none"> <li>• “Implement Component Behavior Using Simulink”</li> <li>• “Create Architecture Reference”</li> </ul>
parameter	A parameter is an instance-specific value of a value type.	Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.	<ul style="list-style-type: none"> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> <li>• “Access Model Arguments as Parameters on Reference Components”</li> <li>• “Use Parameters to Store Instance Values with Components”</li> </ul>

Term	Definition	Application	More Information
subsystem component	A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.	Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Create Simulink Subsystem Behavior Using Subsystem Component”</li> <li>• “Create Simulink Subsystem Component”</li> </ul>
state chart	A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.	Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.	<ul style="list-style-type: none"> <li>• “Implement Behaviors for Architecture Model Simulation”</li> <li>• “Implement Component Behavior Using Stateflow Charts”</li> </ul>

## Version History

Introduced in R2022a

### See Also

addParameter | getParameter | resetToDefault | getParameterPromotedFrom | getEvaluatedParameterValue | getParameterNames | getParameterValue | setParameterValue | resetParameterToDefault

### Topics

“Author Parameters in System Composer Using Parameter Editor”  
 “Access Model Arguments as Parameters on Reference Components”  
 “Use Parameters to Store Instance Values with Components”



# setUnits

**Package:** systemcomposer

Set units for value type

## Syntax

```
setUnits(valueType, units)
```

## Description

setUnits(valueType, units) sets the units for the designated value type.

## Examples

### Set Units for Value Type

Create a model archModel.

```
modelName = "archModel";  
arch = systemcomposer.createModel(modelName, true);
```

Add a value type airSpeed to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType("airSpeed");
```

Set the units for the value type as m/s.

```
airSpeedType.setUnits("m/s")
```

## Input Arguments

### valueType — Value type, data element, or function argument

value type object | data element object | function argument object

Value type, data element, or function argument, specified as a `systemcomposer.ValueType`, `systemcomposer.interface.DataElement`, or `systemcomposer.interface.FunctionArgument` object.

### units — Units

character vector | string

Units, specified as a character vector or string.

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• “Manage Interfaces with Data Dictionaries”</li> <li>• “Reference Data Dictionaries”</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Define Port Interfaces Between Components”</li> </ul>
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2021b

### See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## setValue

**Package:** systemcomposer.analysis

Set value of property for element instance

### Syntax

```
setValue(instance,property,value)
```

### Description

setValue(instance,property,value) sets the property property of the instance instance to the value specified by value.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The instance refers to the element instance on which the iteration is being performed.

---

### Examples

#### Set Mass Property Value

Load the small unmanned aerial vehicle (UAV) model, create an architecture instance, and set the mass property value of a nested component. Get the new value to confirm the change.

```
scExampleSmallUAV
model = systemcomposer.loadModel("scExampleSmallUAVModel");
instance = instantiate(model.Architecture,"UAVComponent","NewInstance");
setValue(instance.Components(1).Components(1),...
"UAVComponent.OnboardElement.Mass",2);
[massValue,unit] = getValue(instance.Components(1).Components(1),...
"UAVComponent.OnboardElement.Mass")
```

```
massValue = 2
```

```
unit =
'kg'
```

### Input Arguments

#### instance — Element instance

architecture instance | component instance | port instance | connector instance

Element instance, specified as a systemcomposer.analysis.ArchitectureInstance, systemcomposer.analysis.ComponentInstance, systemcomposer.analysis.PortInstance, or systemcomposer.analysis.ConnectorInstance object.

**property – Property**

character vector | string

Property, specified in the form "<profile>.<stereotype>.<property>".

Data Types: char | string

**value – Property value**

double (default) | single | int64 | int32 | int16 | int8 | uint64 | uint32 | uint8 | boolean | string | enumeration class name

Property value, specified as a data type that depends on how the property is defined in the profile.

**More About****Definitions**

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	“Run Analysis Function”
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	“Create a Model Instance for Analysis”

Term	Definition	Application	More Information
stereotype	A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.	Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.	“Extend Architectural Design Using Stereotypes”
property	A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.	Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the <b>Property Inspector</b> .	<ul style="list-style-type: none"> <li>• “Set Properties”</li> <li>• “Add Properties with Stereotypes”</li> <li>• “Set Properties for Analysis”</li> </ul>
profile	A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.	Author profiles and apply profiles to a model using the <b>Profile Editor</b> . You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.	<ul style="list-style-type: none"> <li>• “Define Profiles and Stereotypes”</li> <li>• “Use Stereotypes and Profiles”</li> </ul>

## Version History

Introduced in R2019a

### See Also

getValue | hasValue | systemcomposer.analysis.Instance

### Topics

“Write Analysis Function”

“Modeling System Architecture of Small UAV”

## synchronizeChanges

**Package:** systemcomposer.allocation

Synchronize changes of models in allocation set

### Syntax

```
synchronizeChanges(allocSet)
```

### Description

`synchronizeChanges(allocSet)` synchronizes any changes that have been made in the source or target models of the allocation set.

### Examples

#### Synchronize Changes from Models in Allocation Set

This example shows how to synchronize changes for models used in an allocation set.

Create two new models with a component each.

```
mSource = systemcomposer.createModel('Source_Model_Allocation',true);
sourceComp = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');
```

Create the allocation set with name MyAllocation.

```
allocSet = systemcomposer.allocation.createAllocationSet('MyAllocation',...
    'Source_Model_Allocation','Target_Model_Allocation');
```

Get the default allocation scenario.

```
defaultScenario = allocSet.getScenario('Scenario 1');
```

Allocate components between models.

```
allocation = defaultScenario.allocate(sourceComp,targetComp);
```

Update the models with new components.

```
sourceComp2 = mSource.Architecture.addComponent('Source_Component_2');
targetComp2 = mTarget.Architecture.addComponent('Target_Component_2');
```

Synchronize changes from models in allocation set

```
synchronizeChanges(allocSet)
```

Allocate new components between models

```
allocation2 = defaultScenario.allocate(sourceComp2,targetComp2);
```



Open the allocation editor.

```
systemcomposer.allocation.editor
```

Arrange the models so the components appear on the canvas.

```
Simulink.BlockDiagram.arrangeSystem('Source_Model_Allocation')
Simulink.BlockDiagram.arrangeSystem('Target_Model_Allocation')
```

Save the models and allocation set.

```
save(mSource)
save(mTarget)
save(allocSet)
```

## Input Arguments

### **allocSet** — Allocation set

allocation set object

Allocation set, specified as a `systemcomposer.allocation.AllocationSet` object.

## More About

### Definitions

Term	Definition	Application	More Information
allocation	An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.	Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.	<ul style="list-style-type: none"> <li>“Create and Manage Allocations Interactively”</li> <li>“Create and Manage Allocations Programmatically”</li> </ul>
allocation scenario	An allocation scenario contains a set of allocations between a source and a target model.	Allocate between model elements in an allocation scenario. The default allocation scenario is called <b>Scenario 1</b> .	“Systems Engineering Approach for SoC Applications”
allocation set	An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.	Create an allocation set with allocation scenarios in the <b>Allocation Editor</b> . Allocation sets are saved as MLDATX files.	<ul style="list-style-type: none"> <li>“Establish Traceability Between Architectures and Requirements”</li> <li>“Allocate Architectures in Tire Pressure Monitoring System”</li> </ul>

## Version History

Introduced in R2020b

**See Also**

`createScenario` | `deleteScenario` | `getScenario` | `load` |  
`systemcomposer.allocation.AllocationSet.find` | `closeAll` | `close`

**Topics**

“Create and Manage Allocations Programmatically”

# unlinkDictionary

**Package:** systemcomposer.arch

Unlink data dictionary from architecture model

## Syntax

```
unlinkDictionary(model)
```

## Description

unlinkDictionary(model) removes the association of the model from its data dictionary.

## Examples

### Unlink Data Dictionary

Unlink a data dictionary from a model.

```
model = systemcomposer.createModel("newModel", true);  
dictionary = systemcomposer.createDictionary("newDictionary.sldd");  
linkDictionary(model, "newDictionary.sldd")  
save(dictionary)  
save(model)  
unlinkDictionary(model)
```

## Input Arguments

**model** — Architecture model

model object

Architecture model, specified as a systemcomposer.arch.Model object.

## More About

### Definitions

Term	Definition	Application	More Information
architecture	A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.	Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the <b>Parameter Editor</b> .	<ul style="list-style-type: none"> <li>• “Compose Architectures Visually”</li> <li>• “Author Parameters in System Composer Using Parameter Editor”</li> </ul>
model	A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.	Perform operations on a model: <ul style="list-style-type: none"> <li>• Extract the root-level architecture contained in the model.</li> <li>• Apply profiles.</li> <li>• Link interface data dictionaries.</li> <li>• Generate instances from model architecture.</li> </ul> A System Composer model is stored as an SLX file.	“Create Architecture Model with Interfaces and Requirement Links”
component	A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.	Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with: <ul style="list-style-type: none"> <li>• Port interfaces using the <b>Interface Editor</b></li> <li>• Parameters using the <b>Parameter Editor</b></li> </ul>	“Components”

Term	Definition	Application	More Information
port	A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.	There are different types of ports: <ul style="list-style-type: none"> <li>• <i>Component ports</i> are interaction points on the component to other components.</li> <li>• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</li> </ul>	"Ports"
connector	Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.	A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.	"Connections"

Term	Definition	Application	More Information
interface data dictionary	An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.	Local interfaces on a System Composer model can be saved in an interface data dictionary using the <b>Interface Editor</b> . You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> <li>• "Manage Interfaces with Data Dictionaries"</li> <li>• "Reference Data Dictionaries"</li> </ul>
data interface	A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.	Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the <b>Interface Editor</b> to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	<ul style="list-style-type: none"> <li>• "Create Architecture Model with Interfaces and Requirement Links"</li> <li>• "Define Port Interfaces Between Components"</li> </ul>

Term	Definition	Application	More Information
data element	A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Data interfaces are decomposed into data elements: <ul style="list-style-type: none"> <li>• Pins or wires in a connector or harness.</li> <li>• Messages transmitted across a bus.</li> <li>• Data structures shared between components.</li> </ul>	<ul style="list-style-type: none"> <li>• “Create Interfaces”</li> <li>• “Assign Interfaces to Ports”</li> </ul>
value type	A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.	You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the <b>Interface Editor</b> so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.	Create an owned interface to represent a value type or data interface that is local to a port.	“Define Owned Interfaces Local to Ports”

Term	Definition	Application	More Information
adapter	An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the “Interface Adapter” dialog box:</p> <ul style="list-style-type: none"> <li>• Create and edit mappings between input and output interfaces.</li> <li>• Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop.</li> <li>• Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models.</li> <li>• Apply an interface conversion <code>Merge</code> to merges two or more message or signal lines.</li> <li>• When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.</li> </ul>	<ul style="list-style-type: none"> <li>• “Interface Adapter”</li> <li>• Adapter</li> </ul>

## Version History

Introduced in R2019a

### See Also

[linkDictionary](#) | [saveToDictionary](#) | [createDictionary](#) | [addReference](#) | [removeReference](#)

### Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

## update

**Package:** systemcomposer.analysis

Update architecture model

### Syntax

```
update(instance)
```

### Description

`update(instance)` updates a specification model to mirror the changes in the architecture instance. The update method is part of the `systemcomposer.analysis.ArchitectureInstance` class.

---

**Note** This function is part of the instance programmatic interfaces that you can use to analyze the model iteratively, element-by-element. The `instance` refers to the element instance on which the iteration is being performed.

---

## Examples

### Update Specification Model

Update the specification model to mirror the changes in the architecture instance.

Create a profile for latency characteristics and save it.

```
profile = systemcomposer.profile.Profile.createProfile("LatencyProfile");

latencybase = profile.addStereotype("LatencyBase");
latencybase.addProperty("latency", Type="double");
latencybase.addProperty("dataRate", Type="double", DefaultValue="10");

connLatency = profile.addStereotype("ConnectorLatency", ...
Parent="LatencyProfile.LatencyBase");
connLatency.addProperty("secure", Type="boolean");
connLatency.addProperty("linkDistance", Type="double");

nodeLatency = profile.addStereotype("NodeLatency", ...
Parent="LatencyProfile.LatencyBase");
nodeLatency.addProperty("resources", Type="double", DefaultValue="1");

portLatency = profile.addStereotype("PortLatency", ...
Parent="LatencyProfile.LatencyBase");
portLatency.addProperty("queueDepth", Type="double");
portLatency.addProperty("dummy", Type="int32");

profile.save
```

Create a new model. Apply the profile to the model. Apply the stereotype to the architecture. Instantiate all stereotypes in a profile.

```
model = systemcomposer.createModel("archModel", true);
model.applyProfile("LatencyProfile");
```



```
model.Architecture.applyStereotype("LatencyProfile.LatencyBase");
instance = instantiate(model.Architecture,"LatencyProfile","NewInstance");
```

Set a new value for the "dataRate" property on the architecture instance.

```
instance.setValue("LatencyProfile.LatencyBase.dataRate",5);
```

Update the specification model according to the architecture instance.

```
instance.update
```

Get the new value of the "dataRate" property on the architecture.

```
value = model.Architecture.getPropertyValue("LatencyProfile.LatencyBase.dataRate")
```

```
value =
```

```
    '5'
```

## Input Arguments

### **instance** – Architecture instance

architecture instance object

Architecture instance for which specification model is updated, specified as a `systemcomposer.analysis.ArchitectureInstance` object.

## More About

### Definitions

Term	Definition	Application	More Information
analysis	Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.	Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	<ul style="list-style-type: none"> <li>“Analyze Architecture Model with Analysis Function”</li> <li>“Analyze Architecture”</li> <li>“Simple Roll-Up Analysis Using Robot System with Properties”</li> </ul>
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	<ul style="list-style-type: none"> <li>“Analysis Function Constructs”</li> <li>“Write Analysis Function”</li> </ul>

Term	Definition	Application	More Information
instance model	An instance model is a collection of instances.	You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.	"Run Analysis Function"
instance	An instance is an occurrence of an architecture model element at a given point in time.	An instance freezes the active variant or model reference of the component in the instance model.	"Create a Model Instance for Analysis"

## Version History

Introduced in R2019a

### See Also

instantiate | systemcomposer.analysis.Instance | loadInstance | deleteInstance | save | lookup | iterate | refresh

### Topics

"Write Analysis Function"

# systemcomposer.updateLinksToReferenceRequirements

Update requirement links to model reference requirements

## Syntax

```
systemcomposer.updateLinksToReferenceRequirements(modelName, linkDomain,  
documentPathOrID)
```

## Description

`systemcomposer.updateLinksToReferenceRequirements(modelName, linkDomain, documentPathOrID)` imports the external requirement document into Requirements Toolbox as a reference requirement and updates the requirement links to point to the imported set. You can use the `systemcomposer.updateLinksToReferenceRequirements` function in System Composer to make the requirement links point to imported referenced requirements instead of external documents.

## Examples

### Update Reference Requirement Links from Imported File

After importing requirement links from a file, update links to reference requirements for the model. When you convert the links to reference requirement links, the links are contained in the model in an SLREQX file to make full use of Requirements Toolbox™ functionality.

### Import Requirement Links from Word File

Before running the code, follow these steps to prepare your workspace.

**Note:** Importing or linking requirements from an imported file is only supported on Windows. A web-based Microsoft® Office file stored in SharePoint or OneDrive will not work. Save this file and supporting files in a local folder to continue.

1. Open the Microsoft Word file `Functional_Requirements.docx` with the requirements listed. Highlight the requirement to link. For example, highlight these lines.

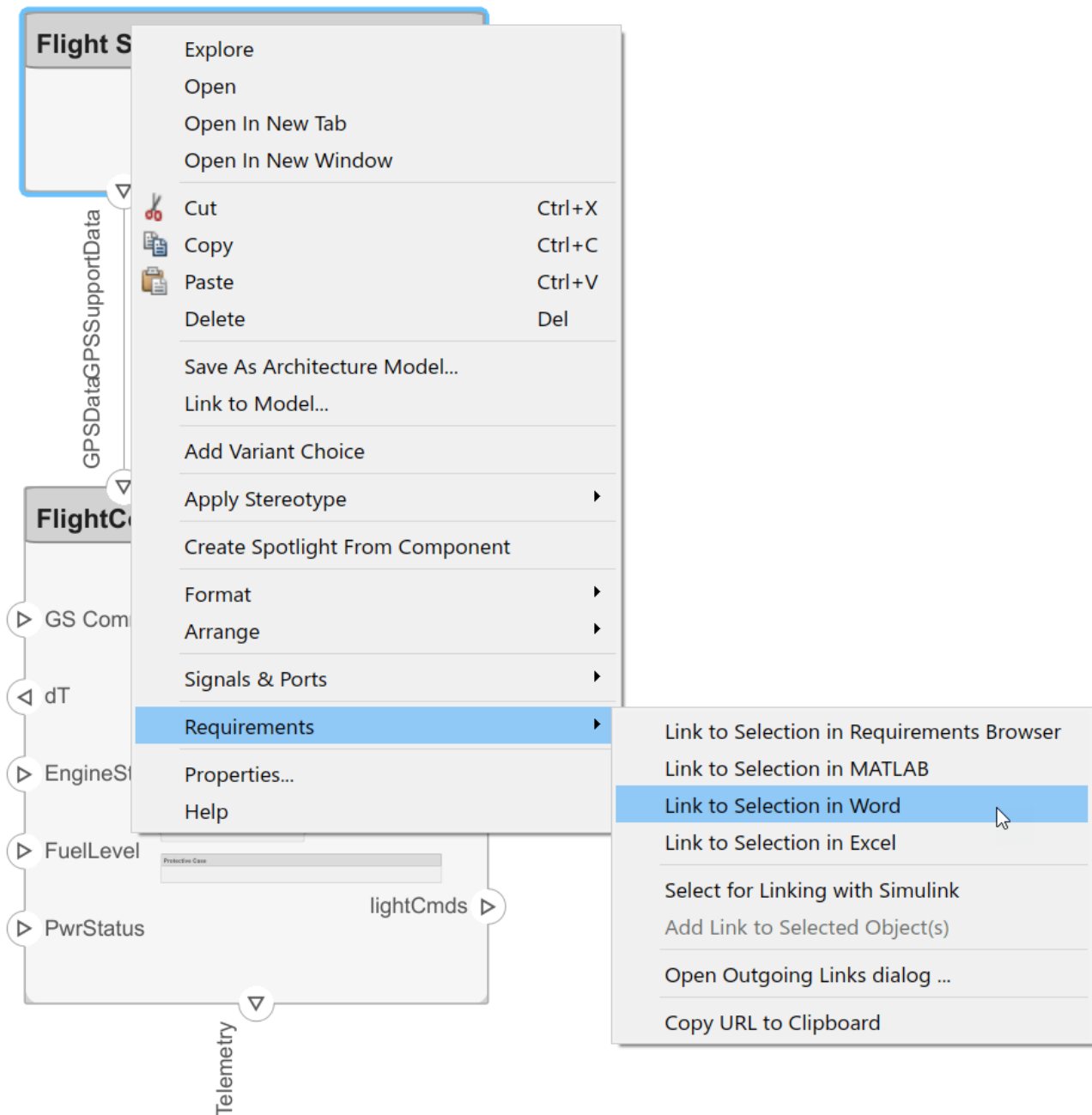
```
1.1.2 Flight Computer  
ID: 25
```

```
Description: Aircraft shall provide a flight computer to autonomously conduct safe flight operat
```

2. Open the `reqImportExample.slx` model.

```
model = systemcomposer.openModel("reqImportExample");
```

3. In the model, select the component to which to link the requirement. Right-click the component and select **Requirements > Link to Selection in Word**. Keep the Word file `Functional_Requirements.docx` open for the next steps.



Before requirement links are integrated within the model, the links depend on the source document, the Word file `Functional_Requirements.docx`. To view the requirement links, open the Requirements Perspective from the bottom-right corner of the `reqImportExample.slx` model palette.

Label	Source	Type	Destination
reqImportExample~mdl.slmx			
Flight Computer ID: 25 Description: Aircraft ...	Flight Support Components	Implements	@Simulink_requirement_item_3

## Update Links to Reference Requirements

Use these steps to update requirement links to integrate with and be referenced from within the model.

1. Export the reqImportExample.slx model and save to an external file: exportedModel.xls

```
exportedSet = systemcomposer.exportModel("reqImportExample");
SaveToExcel("exportedModel",exportedSet);
```

2. Use the external file exportedModel.xls to import requirement links into another model: reqNewExample.slx

```
structModel = ImportModelFromExcel("exportedModel.xls", "Components", "Ports", ...
"Connections", "PortInterfaces", "RequirementLinks");
structModel.readTableFromExcel
```

```
systemcomposer.importModel("reqNewExample",structModel.Components, ...
structModel.Ports,structModel.Connections,structModel.Interfaces,structModel.RequirementLinks);
```

3. To integrate the requirement links to the new model reqNewExample.slx, update references within the model.

```
systemcomposer.updateLinksToReferenceRequirements("reqNewExample", "linktype_rmi_word", "Functiona
```

4. Open the Requirements Perspective from the bottom right corner of the model palette to view the new requirement by setting **View** to **Requirements**.

The screenshot displays a software development environment with three main panels:

- Top Panel:** A diagram showing a **FlightComputer** component with sub-components like **AirData**, **EngineStatus**, **FuelLevel**, **GPSData**, **GS Commands**, and **PwrStatus**. It is connected to **Flight Support Components**, which includes **ADSBData** and **GPSSupport**.
- Bottom Panel (Requirements - reqNewExample):** A table showing a list of requirements. The selected requirement is:
 

Index	ID	Summary
Functional_Requirements		
Import1	Fu...	References to Functional_Requirements.docx
1	1 A...	1 Aircraft Capabilities
- Right Panel (Property Inspector):** Shows the properties for the selected requirement:
  - Requirement: 1 Aircraft Capabilities
  - Type: Functional
  - Index: 1
  - Custom ID: 1 Aircraft Capabilities
  - Summary: 1 Aircraft Capabilities
  - Description: **1 Aircraft Capabilities**
  - Revision information:
    - SID: 2
    - Revision: 1
    - Updated on: 12-Sep-2022 16:39:11
    - Created by:
    - Created on: 12-Sep-2022 16:39:12
    - Modified by:
    - Modified on: 12-Sep-2022 16:22:08

This requirement is saved in a requirement set `Functional_Requirements.slreqx` and used directly in the requirement link. Change the **View** to **Links** to view the requirement link.

Label	Source	Type	Destination
reqNewExample~mdl.slmx	Changed source: 0/1		Changed destination: 0/1
Flight Computer ID: 25 Descriptio...	Flight Support Components	Implements	Simulink_requirement_item_3 Fli...

## Input Arguments

### **modelName** — Name of model

character vector | string

Name of model, specified as a character vector or string.

Example: "exMobileRobot"

Data Types: char | string

### **linkDomain** — Link domain

character vector | string

Link domain, specified as a character vector or string. See "Custom Link Types" (Requirements Toolbox) for more information on identifying your link type or generating custom link types.

Example: "linktype\_rmi\_word"

Data Types: char | string

### **documentPathOrID** — Full document path

character vector | string

Full document path, specified as a character vector or string.

Example: "Functional\_Requirements.docx"

Data Types: char | string

## More About

### Definitions

Term	Definition	Application	More Information
requirements	Requirements are a collection of statements describing the desired behavior and characteristics of a system. Requirements ensure system design integrity and are achievable, verifiable, unambiguous, and consistent with each other. Each level of design should have appropriate requirements.	To enhance traceability of requirements, link system, functional, customer, performance, or design requirements to components and ports. Link requirements to each other to represent derived or allocated requirements. Manage requirements from the Requirements Perspective on an architecture model or through custom views. Assign test cases to requirements using the <b>Test Manager</b> for verification and validation.	<ul style="list-style-type: none"> <li>• “Link and Trace Requirements”</li> <li>• “Establish Traceability Between Architectures and Requirements”</li> </ul>
requirement set	A requirement set is a collection of requirements. You can structure the requirements hierarchically and link them to components or ports.	Use the <b>Requirements Editor</b> to edit and refine requirements in a requirement set. Requirement sets are stored in SLREQX files. You can create a new requirement set and author requirements using Requirements Toolbox, or import requirements from supported third-party tools.	<ul style="list-style-type: none"> <li>• “Allocate and Trace Requirements from Design to Verification”</li> <li>• “Manage Requirements”</li> </ul>
requirement link	A link is an object that relates two model-based design elements. A requirement link is a link where the destination is a requirement. You can link requirements to components or ports.	View links using the Requirements Perspective in System Composer. Select a requirement in the Requirements Browser to highlight the component or the port to which the requirement is assigned. Links are stored externally as SLMX files.	<ul style="list-style-type: none"> <li>• “Create Architecture Model with Interfaces and Requirement Links”</li> <li>• “Update Reference Requirement Links from Imported File” on page 4-791</li> </ul>



Term	Definition	Application	More Information
test harness	A test harness is a model that isolates the component under test with inputs, outputs, and verification blocks configured for testing scenarios. You can create a test harness for a model component or for a full model. A test harness gives you a separate testing environment for a model or a model component.	Create a test harness for a System Composer component to validate simulation results and verify design. To edit the interfaces while you are testing the behavior of a component in a test harness, use the <b>Interface Editor</b> .	<ul style="list-style-type: none"> <li>“Verify and Validate Requirements”</li> <li>“Create a Test Harness” (Simulink Test)</li> </ul>

## Version History

Introduced in R2020b

### See Also

importModel | exportModel

### Topics

“Allocate and Trace Requirements from Design to Verification”

“Import and Export Architecture Models”

“Custom Link Types” (Requirements Toolbox)



# Methods

---

## find

**Class:** `systemcomposer.rptgen.finder.AllocationListFinder`

**Package:** `systemcomposer.rptgen.finder`

Find allocations to and from component

### Syntax

```
result = find(finder)
```

### Description

`result = find(finder)` finds allocations to or from a particular component for the `AllocationList` search result.

### Input Arguments

**finder** — Allocation list finder

allocation list finder object

Allocation list finder, specified as a `systemcomposer.rptgen.finder.AllocationListFinder` object.

### Output Arguments

**result** — Allocation list result

allocation list result object

Allocation list result, returned as a `systemcomposer.rptgen.finder.AllocationListResult` object.

## Examples

### Generate AllocationList Result Report

Use the `AllocationListFinder` and `AllocationListResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationListResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocations"));
add(rpt,TableOfContents);

allocationListFinder = AllocationListFinder("AllocationSet.mldatx");
allocationListFinder.ComponentName = "mTestModel/Component1";
```

```
chapter = Chapter("Title",allocationListFinder.ComponentName);
result = find(allocationListFinder);
reporter = getReporter(result);

add(rpt,chapter);
append(rpt,reporter);
close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationListFinder |  
systemcomposer.rptgen.finder.AllocationListResult |  
systemcomposer.rptgen.report.AllocationList | next | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.AllocationListFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if allocation list search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the `AllocationList` search result queue is nonempty.

### Input Arguments

**finder — Allocation list finder**

allocation list finder object

Allocation list finder, specified as a `systemcomposer.rptgen.finder.AllocationListFinder` object.

### Output Arguments

**nonempty — Whether queue is nonempty**

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: logical

## Examples

### Generate AllocationList Finder Report

Use the `AllocationListFinder` and `AllocationListResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationListFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocations"));
add(rpt,TableOfContents);

allocationListFinder = AllocationListFinder("AllocationSet.mldatx");
allocationListFinder.ComponentName = "mTestModel/Component1";
```

```
chapter = Chapter("Title","Allocations");
while hasNext(allocationListFinder)
    allocations = next(allocationListFinder);
    sect = Section("Title",allocationListFinder.ComponentName);
    add(sect,allocations);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationListFinder |  
systemcomposer.rptgen.finder.AllocationListResult |  
systemcomposer.rptgen.report.AllocationList | find | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.AllocationListFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next allocation list search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next `AllocationList` search result.

### Input Arguments

**finder** — Allocation list finder

allocation list finder object

Allocation list finder, specified as a `systemcomposer.rptgen.finder.AllocationListFinder` object.

### Output Arguments

**result** — Allocation list result

allocation list result object

Allocation list result, returned as a `systemcomposer.rptgen.finder.AllocationListResult` object.

## Examples

### Generate AllocationList Finder Report

Use the `AllocationListFinder` and `AllocationListResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationListFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocations"));
add(rpt,TableOfContents);

allocationListFinder = AllocationListFinder("AllocationSet.mldatx");
allocationListFinder.ComponentName = "mTestModel/Component1";
chapter = Chapter("Title","Allocations");
while hasNext(allocationListFinder)
```



```
        allocations = next(allocationListFinder);
        sect = Section("Title",allocationListFinder.ComponentName);
        add(sect,allocations);
        add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationListFinder |  
systemcomposer.rptgen.finder.AllocationListResult |  
systemcomposer.rptgen.report.AllocationList | find | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.AllocationListResult`

**Package:** `systemcomposer.rptgen.finder`

Get allocation list reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about allocations in a component. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.AllocationList` reporter class for more information on how to customize the reporter.

### Input Arguments

**result — Allocation list result**

allocation list result object

Allocation list result, specified as a `systemcomposer.rptgen.finder.AllocationListResult` object.

### Output Arguments

**reporter — Allocation list reporter**

allocation list reporter object

Allocation list reporter, returned as a `systemcomposer.rptgen.report.AllocationList` object.

### Examples

#### Generate AllocationList Result Report

Use the `AllocationListFinder` and `AllocationListResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationListResultReport", ...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocations"));
add(rpt,TableOfContents);

allocationListFinder = AllocationListFinder("AllocationSet.mldatx");
```

```
allocationListFinder.ComponentName = "mTestModel/Component1";
chapter = Chapter("Title",allocationListFinder.ComponentName);
result = find(allocationListFinder);
reporter = getReporter(result);

add(rpt,chapter);
append(rpt,reporter);
close(rpt);
rptview(rpt)
```

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.AllocationListFinder |  
systemcomposer.rptgen.finder.AllocationListResult |  
systemcomposer.rptgen.report.AllocationList | find | next | hasNext |  
createTemplate | customizeReporter | getClassFolder

### Topics

"System Composer Report Generation for System Architectures"  
"System Composer Report Generation for Software Architectures"

## find

**Class:** `systemcomposer.rptgen.finder.AllocationSetFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about allocation set

### Syntax

```
result = find(finder)
```

### Description

`result = find(finder)` finds information about the allocation set for the `AllocationSet` search result.

### Input Arguments

**finder** — Allocation set finder

allocation set finder object

Allocation set finder, specified as a `systemcomposer.rptgen.finder.AllocationSetFinder` object.

### Output Arguments

**result** — Allocation set result

allocation set result object

Allocation set result, returned as a `systemcomposer.rptgen.finder.AllocationSetResult` object.

## Examples

### Generate AllocationSet Result Report

Use the `AllocationSetFinder` and `AllocationSetResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationSetResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocation Sets"));
add(rpt,TableOfContents);
chapter = Chapter("Title","Allocation Sets");

allocationSetFinder = AllocationSetFinder("AllocationSet.mldatx");
```

```
result = find(allocationSetFinder);
reporter = getReporter(result);

add(rpt,chapter);
append(rpt,reporter);
close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationSetFinder |  
systemcomposer.rptgen.finder.AllocationSetResult |  
systemcomposer.rptgen.report.AllocationSet | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.AllocationSetFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if allocation set search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the `AllocationSet` search result queue is nonempty.

### Input Arguments

**finder — Allocation set finder**

allocation set finder object

Allocation set finder, specified as a `systemcomposer.rptgen.finder.AllocationSetFinder` object.

### Output Arguments

**nonempty — Whether queue is nonempty**

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: logical

## Examples

### Generate AllocationSet Finder Report

Use the `AllocationSetFinder` and `AllocationSetResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationSetFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocation Sets"));
add(rpt,TableOfContents);

allocationSetFinder = AllocationSetFinder("AllocationSet.mldatx");
chapter = Chapter("Title","Allocation Set");
```

```
while hasNext(allocationSetFinder)
  allocationSets = next(allocationSetFinder);
  sect = Section(strcat("Allocations in ",allocationSets.Name));
  add(sect,allocationSets);
  add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationSetFinder |  
systemcomposer.rptgen.finder.AllocationSetResult |  
systemcomposer.rptgen.report.AllocationSet | find | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.AllocationSetFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next allocation set search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next `AllocationSet` search result.

### Input Arguments

**finder** — Allocation set finder

allocation set finder object

Allocation set finder, specified as a `systemcomposer.rptgen.finder.AllocationSetFinder` object.

### Output Arguments

**result** — Allocation set result

allocation set result object

Allocation set result, returned as a `systemcomposer.rptgen.finder.AllocationSetResult` object.

## Examples

### Generate AllocationSet Finder Report

Use the `AllocationSetFinder` and `AllocationSetResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationSetFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocation Sets"));
add(rpt,TableOfContents);

allocationSetFinder = AllocationSetFinder("AllocationSet.mldatx");
chapter = Chapter("Title","Allocation Set");

while hasNext(allocationSetFinder)
```



```
allocationSets = next(allocationSetFinder);
sect = Section(strcat("Allocations in ",allocationSets.Name));
add(sect,allocationSets);
add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationSetFinder |  
systemcomposer.rptgen.finder.AllocationSetResult |  
systemcomposer.rptgen.report.AllocationSet | find | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.AllocationSetResult`

**Package:** `systemcomposer.rptgen.finder`

Get allocation set reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about allocation sets in a model. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.AllocationSetReporter` class for more information on how to customize the reporter.

### Input Arguments

**result — Allocation set result**

allocation set result object

Allocation set result, specified as a `systemcomposer.rptgen.finder.AllocationSetResult` object.

### Output Arguments

**reporter — Allocation set reporter**

allocation set reporter object

Allocation set reporter, returned as a `systemcomposer.rptgen.report.AllocationSet` object.

### Examples

#### Generate AllocationSet Result Report

Use the `AllocationSetFinder` and `AllocationSetResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

rpt = slreportgen.report.Report(output="AllocationSetResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Allocation Sets"));
add(rpt,TableOfContents);
chapter = Chapter("Title","Allocation Sets");
```

```
allocationSetFinder = AllocationSetFinder("AllocationSet.mldatx");
result = find(allocationSetFinder);
reporter = getReporter(result);

add(rpt, chapter);
append(rpt, reporter);
close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.AllocationSetFinder |  
systemcomposer.rptgen.finder.AllocationSetResult |  
systemcomposer.rptgen.report.AllocationSet | find | hasNext | next | createTemplate  
| customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## find

**Class:** `systemcomposer.rptgen.finder.ComponentFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about component

### Syntax

```
result = find(finder)
```

### Description

`result = find(finder)` finds information about a component for the Component search result.

### Input Arguments

**finder — Component finder**

component finder object

Component finder, specified as a `systemcomposer.rptgen.finder.ComponentFinder` object.

### Output Arguments

**result — Component result**

component result object | array of component result objects

Component result, returned as a `systemcomposer.rptgen.finder.ComponentResult` object or an array of `systemcomposer.rptgen.finder.ComponentResult` objects.

## Examples

### Generate Component Result Report

Use the `ComponentFinder` and `ComponentResult` classes to generate a report.

```
import systemcomposer.rptgen.finder.*
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.query.*

rpt = slreportgen.report.Report(output="ComponentResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Components"));
add(rpt,TableOfContents);
chapter = Chapter("Title","Components");

componentFinder = ComponentFinder("mTestModel");
componentFinder.Query = AnyComponent;
result = find(componentFinder);
```

```
for i = result
    reporter = getReporter(i);
    reporter.IncludeProperties = false;
    reporter.IncludeSnapshot = false;
    add(chapter, reporter);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ComponentFinder |  
systemcomposer.rptgen.finder.ComponentResult |  
systemcomposer.rptgen.report.Component | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.ComponentFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if component search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the Component search result queue is nonempty.

### Input Arguments

**finder — Component finder**

component finder object

Component finder, specified as a `systemcomposer.rptgen.finder.ComponentFinder` object.

### Output Arguments

**nonempty — Whether queue is nonempty**

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

## Examples

### Generate Component Finder Report

Use the `ComponentFinder` and `ComponentResult` classes to generate a report.

```
import systemcomposer.rptgen.finder.*
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.query.*

rpt = slreportgen.report.Report(output="ComponentFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Components"));
add(rpt,TableOfContents);

componentFinder = ComponentFinder("mTestModel");
componentFinder.Query = AnyComponent;
```

```
chapter = Chapter("Components in mTestModel");

while hasNext(componentFinder)
    componentResult = next(componentFinder);
    sect = Section(componentResult.Name);
    add(sect,componentResult);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ComponentFinder |  
systemcomposer.rptgen.finder.ComponentResult |  
systemcomposer.rptgen.report.Component | find | next | getReporter | createTemplate  
| customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.ComponentFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next component search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next Component search result.

### Input Arguments

**finder** — Component finder

component finder object

Component finder, specified as a `systemcomposer.rptgen.finder.ComponentFinder` object.

### Output Arguments

**result** — Component result

component result object

Component result, returned as a `systemcomposer.rptgen.finder.ComponentResult` object.

## Examples

### Generate Component Finder Report

Use the `ComponentFinder` and `ComponentResult` classes to generate a report.

```
import systemcomposer.rptgen.finder.*
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.query.*

rpt = slreportgen.report.Report(output="ComponentFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Components"));
add(rpt,TableOfContents);

componentFinder = ComponentFinder("mTestModel");
componentFinder.Query = AnyComponent;

chapter = Chapter("Components in mTestModel");
```



```
while hasNext(componentFinder)
    componentResult = next(componentFinder);
    sect = Section(componentResult.Name);
    add(sect,componentResult);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.ComponentFinder |  
systemcomposer.rptgen.finder.ComponentResult |  
systemcomposer.rptgen.report.Component | find | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.ComponentResult`

**Package:** `systemcomposer.rptgen.finder`

Get component reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about components in a model. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.Component` reporter class for more information on how to customize the reporter.

### Input Arguments

**result – Component result**

component result object

Component result, specified as a `systemcomposer.rptgen.finder.ComponentResult` object.

### Output Arguments

**reporter – Component reporter**

component reporter object

Component reporter, returned as a `systemcomposer.rptgen.report.Component` object.

### Examples

#### Generate Component Result Report

Use the `ComponentFinder` and `ComponentResult` classes to generate a report.

```
import systemcomposer.rptgen.finder.*
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.query.*

rpt = slreportgen.report.Report(output="ComponentResultReport", ...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title","Components"));
add(rpt,TableOfContents);
chapter = Chapter("Title","Components");
```

```
componentFinder = ComponentFinder("mTestModel");
componentFinder.Query = AnyComponent;
result = find(componentFinder);

for i = result
    reporter = getReporter(i);
    reporter.IncludeProperties = false;
    reporter.IncludeSnapshot = false;
    add(chapter, reporter);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.ComponentFinder |  
systemcomposer.rptgen.finder.ComponentResult |  
systemcomposer.rptgen.report.Component | find | hasNext | next | createTemplate |  
customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## find

**Class:** `systemcomposer.rptgen.finder.ConnectorFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about connector

### Syntax

```
result = find(finder)
```

### Description

`result = find(finder)` finds information about a connector for the `Connector` search result.

### Input Arguments

**finder** — Connector finder

*connector finder object*

Connector finder, specified as a `systemcomposer.rptgen.finder.ConnectorFinder` object.

### Output Arguments

**result** — Connector result

*connector result object*

Connector result, returned as a `systemcomposer.rptgen.finder.ConnectorResult` object.

## Examples

### Generate Connector Result Report

Use the `ConnectorFinder` and `ConnectorResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ConnectorResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Connectors in %s Model',model_name)));
add(rpt,TableOfContents);

connectorFinder = ConnectorFinder(model_name);
connectorFinder.Filter = "Component";
connectorFinder.ComponentName = "scExampleSmallUAVModel/Flight Support Components";
```

```
chapter = Chapter("Title", "Connectors");
result = find(connectorFinder);
add(rpt, chapter);

for r = result
    reporter = getReporter(r);
    append(rpt, reporter);
end

close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ConnectorFinder |  
systemcomposer.rptgen.finder.ConnectorResult |  
systemcomposer.rptgen.report.Connector | hasNext | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.ConnectorFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if connector search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the Connector search result queue is nonempty.

### Input Arguments

**finder — Connector finder**

connector finder object

Connector finder, specified as a `systemcomposer.rptgen.finder.ConnectorFinder` object.

### Output Arguments

**nonempty — Whether queue is nonempty**

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

## Examples

### Generate Connector Finder Report

Use the `ConnectorFinder` and `ConnectorResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ConnectorFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Connectors in %s Model',model_name)));
add(rpt,TableOfContents);
```

```
connectorFinder = ConnectorFinder(model_name);
connectorFinder.ComponentName = "scExampleSmallUAVModel/Flight Support Components/GPS Module";
connectorFinder.Filter = "Component";
chapter = Chapter("Title", "Connectors");
while hasNext(connectorFinder)
    connector = next(connectorFinder);
    sect = Section("Title", connector.Name);
    add(sect, connector);
    add(chapter, sect);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ConnectorFinder |  
systemcomposer.rptgen.finder.ConnectorResult |  
systemcomposer.rptgen.report.Connector | find | next | getReporter | createTemplate  
| customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.ConnectorFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next connector search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next Connector search result.

### Input Arguments

**finder** — Connector finder

connector finder object

Connector finder, specified as a `systemcomposer.rptgen.finder.ConnectorFinder` object.

### Output Arguments

**result** — Connector result

connector result object

Connector result, returned as a `systemcomposer.rptgen.finder.ConnectorResult` object.

## Examples

### Generate Connector Finder Report

Use the `ConnectorFinder` and `ConnectorResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ConnectorFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Connectors in %s Model',model_name)));
add(rpt,TableOfContents);

connectorFinder = ConnectorFinder(model_name);
connectorFinder.ComponentName = "scExampleSmallUAVModel/Flight Support Components/GPS Module";
connectorFinder.Filter = "Component";
```



```
chapter = Chapter("Title", "Connectors");
while hasNext(connectorFinder)
    connector = next(connectorFinder);
    sect = Section("Title", connector.Name);
    add(sect, connector);
    add(chapter, sect);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ConnectorFinder |  
systemcomposer.rptgen.finder.ConnectorResult |  
systemcomposer.rptgen.report.Connector | find | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.ConnectorResult`

**Package:** `systemcomposer.rptgen.finder`

Get connector reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about connectors in a component. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.ConnectorReporter` class for more information on how to customize the reporter.

### Input Arguments

**result** — Connector result

connector result object

Connector result, specified as a `systemcomposer.rptgen.finder.ConnectorResult` object.

### Output Arguments

**reporter** — Connector reporter

connector reporter object

Connector reporter, returned as a `systemcomposer.rptgen.report.Connector` object.

## Examples

### Generate Connector Result Report

Use the `ConnectorFinder` and `ConnectorResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ConnectorResultReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Connectors in %s Model',model_name)));
add(rpt,TableOfContents);
```

```
connectorFinder = ConnectorFinder(model_name);
connectorFinder.Filter = "Component";
connectorFinder.ComponentName = "scExampleSmallUAVModel/Flight Support Components";
chapter = Chapter("Title", "Connectors");
result = find(connectorFinder);
add(rpt, chapter);

for r = result
    reporter = getReporter(r);
    append(rpt, reporter);
end

close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ConnectorFinder |  
systemcomposer.rptgen.finder.ConnectorResult |  
systemcomposer.rptgen.report.Connector | find | hasNext | next | createTemplate |  
customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## find

**Class:** `systemcomposer.rptgen.finder.DictionaryFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about dictionary

### Syntax

```
result = find(finder)
```

### Description

`result = find(finder)` finds information about a dictionary for the Dictionary search result.

### Input Arguments

**finder** – Dictionary finder

dictionary finder object

Dictionary finder, specified as a `systemcomposer.rptgen.finder.DictionaryFinder` object.

### Output Arguments

**result** – Dictionary result

dictionary result object

Dictionary result, returned as a `systemcomposer.rptgen.finder.DictionaryResult` object.

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.finder.DictionaryFinder` |  
`systemcomposer.rptgen.finder.DictionaryResult` | `hasNext` | `next`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# hasNext

**Class:** `systemcomposer.rptgen.finder.DictionaryFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if dictionary search result queue is nonempty

## Syntax

```
nonempty = hasNext(finder)
```

## Description

`nonempty = hasNext(finder)` determines whether the Dictionary search result queue is nonempty.

## Input Arguments

**finder** — Dictionary finder

dictionary finder object

Dictionary finder, specified as a `systemcomposer.rptgen.finder.DictionaryFinder` object.

## Output Arguments

**nonempty** — Whether queue is nonempty

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: logical

## Examples

### Generate Dictionary Finder Report

Use the `DictionaryFinder` and `DictionaryResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scKeylessEntrySystem
model_name = "KeylessEntryArchitecture";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="DictionaryFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Dictionaries in %s Model',model_name)));
add(rpt,TableOfContents);
```

```
dictFinder = DictionaryFinder(model_name);

chapter = Chapter("Title", "Dictionaries");
while hasNext(dictFinder)
    dict = next(dictFinder);
    sect = Section("Title", dict.Name);
    add(sect, dict);
    add(chapter, sect);
end

add(rpt, chapter);
close(rpt);
rptview(rpt)
```

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.finder.DictionaryFinder` |  
`systemcomposer.rptgen.finder.DictionaryResult` | `find` | `next`

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.DictionaryFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next dictionary search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next Dictionary search result.

### Input Arguments

**finder** — Dictionary finder

dictionary finder object

Dictionary finder, specified as a `systemcomposer.rptgen.finder.DictionaryFinder` object.

### Output Arguments

**result** — Dictionary result

dictionary result object

Dictionary result, returned as a `systemcomposer.rptgen.finder.DictionaryResult` object.

## Examples

### Generate Dictionary Finder Report

Use the `DictionaryFinder` and `DictionaryResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scKeylessEntrySystem
model_name = "KeylessEntryArchitecture";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="DictionaryFinderReport", ...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Dictionaries in %s Model',model_name)));
add(rpt,TableOfContents);

dictFinder = DictionaryFinder(model_name);

chapter = Chapter("Title","Dictionaries");
```

```
while hasNext(dictFinder)
  dict = next(dictFinder);
  sect = Section("Title",dict.Name);
  add(sect,dict);
  add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt)
```

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.DictionaryFinder |  
systemcomposer.rptgen.finder.DictionaryResult | find | hasNext

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”



# find

**Class:** `systemcomposer.rptgen.finder.FunctionFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about function

## Syntax

```
result = find(finder)
```

## Description

`result = find(finder)` finds information about a function for the `Function` search result.

## Input Arguments

**finder** — **Function finder**

function finder object

Function finder, specified as a `systemcomposer.rptgen.finder.FunctionFinder` object.

## Output Arguments

**result** — **Function result**

function result object

Function result, returned as a `systemcomposer.rptgen.finder.FunctionResult` object.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.FunctionFinder` |  
`systemcomposer.rptgen.finder.FunctionResult` |  
`systemcomposer.rptgen.report.Function` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.FunctionFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if function search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the `Function` search result queue is nonempty.

### Input Arguments

**finder** — **Function finder**

function finder object

Function finder, specified as a `systemcomposer.rptgen.finder.FunctionFinder` object.

### Output Arguments

**nonempty** — **Whether queue is nonempty**

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.FunctionFinder` |  
`systemcomposer.rptgen.finder.FunctionResult` |  
`systemcomposer.rptgen.report.Function` | `find` | `next` | `getReporter` | `createTemplate` |  
`customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.FunctionFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next function search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next Function search result.

### Input Arguments

**finder** — Function finder

function finder object

Function finder, specified as a `systemcomposer.rptgen.finder.FunctionFinder` object.

### Output Arguments

**result** — Function result

function result object

Function result, returned as a `systemcomposer.rptgen.finder.FunctionResult` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.FunctionFinder` |  
`systemcomposer.rptgen.finder.FunctionResult` |  
`systemcomposer.rptgen.report.Function` | `find` | `hasNext` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.FunctionResult`

**Package:** `systemcomposer.rptgen.finder`

Get function reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about functions in a software architecture model. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.Function` reporter class for more information on how to customize the reporter.

### Input Arguments

**result** – Function result

function result object

Function result, specified as a `systemcomposer.rptgen.finder.FunctionResult` object.

### Output Arguments

**reporter** – Function reporter

function reporter object

Function reporter, returned as a `systemcomposer.rptgen.report.Function` object.

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.finder.FunctionFinder` |  
`systemcomposer.rptgen.finder.FunctionResult` |  
`systemcomposer.rptgen.report.Function` | `find` | `hasNext` | `next` | `createTemplate` |  
`customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# find

**Class:** `systemcomposer.rptgen.finder.InterfaceFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about interface

## Syntax

```
result = find(finder)
```

## Description

`result = find(finder)` finds information about an interface for the `Interface` search result.

## Input Arguments

**finder — Interface finder**

interface finder object

Interface finder, specified as a `systemcomposer.rptgen.finder.InterfaceFinder` object.

## Output Arguments

**result — Interface result**

interface result object

Interface result, returned as a `systemcomposer.rptgen.finder.InterfaceResult` object.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.InterfaceFinder` |  
`systemcomposer.rptgen.finder.InterfaceResult` |  
`systemcomposer.rptgen.report.Interface` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.InterfaceFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if interface search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the Interface search result queue is nonempty.

### Input Arguments

**finder — Interface finder**

interface finder object

Interface finder, specified as a `systemcomposer.rptgen.finder.InterfaceFinder` object.

### Output Arguments

**nonempty — Whether queue is nonempty**

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

## Examples

### Generate Interface Finder Report

Use the `InterfaceFinder` and `InterfaceResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="InterfaceFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Interfaces in %s Model',model_name)));
add(rpt,TableOfContents);
```

```
intfFinder = InterfaceFinder(model_name);

chapter = Chapter("Title", "Interfaces");
while hasNext(intfFinder)
    interface = next(intfFinder);
    sect = Section("Title", interface.InterfaceName);
    add(sect, interface);
    add(chapter, sect);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.InterfaceFinder |  
systemcomposer.rptgen.finder.InterfaceResult |  
systemcomposer.rptgen.report.Interface | find | next | getReporter | createTemplate  
| customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.InterfaceFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next interface search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next Interface search result.

### Input Arguments

**finder — Interface finder**

interface finder object

Interface finder, specified as a `systemcomposer.rptgen.finder.InterfaceFinder` object.

### Output Arguments

**result — Interface result**

interface result object

Interface result, returned as a `systemcomposer.rptgen.finder.InterfaceResult` object.

## Examples

### Generate Interface Finder Report

Use the `InterfaceFinder` and `InterfaceResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="InterfaceFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Interfaces in %s Model',model_name)));
add(rpt,TableOfContents);

intfFinder = InterfaceFinder(model_name);

chapter = Chapter("Title", "Interfaces");
```



```
while hasNext(intfFinder)
  interface = next(intfFinder);
  sect = Section("Title",interface.InterfaceName);
  add(sect,interface);
  add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.InterfaceFinder |  
systemcomposer.rptgen.finder.InterfaceResult |  
systemcomposer.rptgen.report.Interface | find | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.InterfaceResult`

**Package:** `systemcomposer.rptgen.finder`

Get interface reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about interfaces in a model. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.Interface` reporter class for more information on how to customize the reporter.

### Input Arguments

**result – Interface result**

interface result object

Interface result, specified as a `systemcomposer.rptgen.finder.InterfaceResult` object.

### Output Arguments

**reporter – Interface reporter**

interface reporter object

Interface reporter, returned as a `systemcomposer.rptgen.report.Interface` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.InterfaceFinder` |  
`systemcomposer.rptgen.finder.InterfaceResult` |  
`systemcomposer.rptgen.report.Interface` | `find` | `hasNext` | `next` | `createTemplate` |  
`customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# find

**Class:** `systemcomposer.rptgen.finder.ProfileFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about profile

## Syntax

```
result = find(finder)
```

## Description

`result = find(finder)` finds information about a profile for the Profile search result.

## Input Arguments

**finder** — Profile finder

profile finder object

Profile finder, specified as a `systemcomposer.rptgen.finder.ProfileFinder` object.

## Output Arguments

**result** — Profile result

profile result object

Profile result, returned as a `systemcomposer.rptgen.finder.ProfileResult` object.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.ProfileFinder` |  
`systemcomposer.rptgen.finder.ProfileResult` |  
`systemcomposer.rptgen.report.Profile` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.ProfileFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if profile search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the Profile search result queue is nonempty.

### Input Arguments

**finder** — Profile finder

profile finder object

Profile finder, specified as a `systemcomposer.rptgen.finder.ProfileFinder` object.

### Output Arguments

**nonempty** — Whether queue is nonempty

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

## Examples

### Generate Profile Finder Report

Use the `ProfileFinder` and `ProfileResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ProfileFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Profiles in %s Model',model_name)));
add(rpt,TableOfContents);

profileFinder = ProfileFinder("UAVComponent");
```

```
chapter = Chapter("Title", "Profiles");
while hasNext(profileFinder)
    profile = next(profileFinder);
    sect = Section("Title", profile.Name);
    add(sect, profile);
    add(chapter, sect);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ProfileFinder |  
systemcomposer.rptgen.finder.ProfileResult |  
systemcomposer.rptgen.report.Profile | find | next | getReporter | createTemplate |  
customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.ProfileFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next profile search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next Profile search result.

### Input Arguments

**finder** — Profile finder

profile finder object

Profile finder, specified as a `systemcomposer.rptgen.finder.ProfileFinder` object.

### Output Arguments

**result** — Profile result

profile result object

Profile result, returned as a `systemcomposer.rptgen.finder.ProfileResult` object.

## Examples

### Generate Profile Finder Report

Use the `ProfileFinder` and `ProfileResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ProfileFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Profiles in %s Model',model_name)));
add(rpt,TableOfContents);

profileFinder = ProfileFinder("UAVComponent");

chapter = Chapter("Title","Profiles");
```

```
while hasNext(profileFinder)
  profile = next(profileFinder);
  sect = Section("Title",profile.Name);
  add(sect,profile);
  add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ProfileFinder |  
systemcomposer.rptgen.finder.ProfileResult |  
systemcomposer.rptgen.report.Profile | find | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.ProfileResult`

**Package:** `systemcomposer.rptgen.finder`

Get profile reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about profiles in a model. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.Profile` reporter class for more information on how to customize the reporter.

### Input Arguments

**result — Profile result**

profile result object

Profile result, specified as a `systemcomposer.rptgen.finder.ProfileResult` object.

### Output Arguments

**reporter — Profile reporter**

profile reporter object

Profile reporter, returned as a `systemcomposer.rptgen.report.Profile` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.ProfileFinder` |  
`systemcomposer.rptgen.finder.ProfileResult` |  
`systemcomposer.rptgen.report.Profile` | `find` | `hasNext` | `next` | `createTemplate` |  
`customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# find

**Class:** `systemcomposer.rptgen.finder.RequirementLinkFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about requirement link

## Syntax

```
result = find(finder)
```

## Description

`result = find(finder)` finds information about a requirement link for the `RequirementLink` search result.

## Input Arguments

**finder — Requirement link finder**

requirement link finder object

Requirement link finder, specified as a `systemcomposer.rptgen.finder.RequirementLinkFinder` object.

## Output Arguments

**result — Requirement link result**

requirement link result object

Requirement link result, returned as a `systemcomposer.rptgen.finder.RequirementLinkFinder` object.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.RequirementLinkFinder` |  
`systemcomposer.rptgen.finder.RequirementLinkResult` |  
`systemcomposer.rptgen.report.RequirementLink` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.RequirementLinkFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if requirement link search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the `RequirementLink` search result queue is nonempty.

### Input Arguments

**finder — Requirement link finder**

requirement link finder object

Requirement link finder, specified as a `systemcomposer.rptgen.finder.RequirementLinkFinder` object.

### Output Arguments

**nonempty — Whether queue is nonempty**

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.RequirementLinkFinder` |  
`systemcomposer.rptgen.finder.RequirementLinkResult` |  
`systemcomposer.rptgen.report.RequirementLink` | `find` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.RequirementLinkFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next requirement link search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next `RequirementLink` search result.

### Input Arguments

**finder** — Requirement link finder

requirement link finder object

Requirement link finder, specified as a `systemcomposer.rptgen.finder.RequirementLinkFinder` object.

### Output Arguments

**result** — Requirement link result

requirement link result object

Requirement link result, returned as a `systemcomposer.rptgen.finder.RequirementLinkFinder` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.RequirementLinkFinder` |  
`systemcomposer.rptgen.finder.RequirementLinkResult` |  
`systemcomposer.rptgen.report.RequirementLink` | `find` | `hasNext` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.RequirementLinkResult`

**Package:** `systemcomposer.rptgen.finder`

Get requirement links reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about requirement links in a requirement link set. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.RequirementLink` reporter class for more information on how to customize the reporter.

### Input Arguments

**result** — Requirement link result

requirement link result object

Requirement link result, specified as a `systemcomposer.rptgen.finder.RequirementLinkResult` object.

### Output Arguments

**reporter** — Requirement link reporter

requirement link reporter object

Requirement link reporter, returned as a `systemcomposer.rptgen.report.RequirementLink` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.RequirementLinkFinder` |  
`systemcomposer.rptgen.finder.RequirementLinkResult` |  
`systemcomposer.rptgen.report.RequirementLink` | `find` | `hasNext` | `next` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# find

**Class:** `systemcomposer.rptgen.finder.RequirementSetFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about requirement

## Syntax

```
result = find(finder)
```

## Description

`result = find(finder)` finds information about a requirement for the RequirementSet search result.

## Input Arguments

**finder — Requirement set finder**

requirement set finder object

Requirement set finder, specified as a `systemcomposer.rptgen.finder.RequirementSetFinder` object.

## Output Arguments

**result — Requirement set result**

requirement set result object

Requirement set result, returned as a `systemcomposer.rptgen.finder.RequirementSetResult` object.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.RequirementSetFinder` |  
`systemcomposer.rptgen.finder.RequirementSetResult` |  
`systemcomposer.rptgen.report.RequirementSet` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.RequirementSetFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if requirement set search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the `RequirementSet` search result queue is nonempty.

### Input Arguments

**finder — Requirement set finder**

requirement set finder object

Requirement set finder, specified as a `systemcomposer.rptgen.finder.RequirementSetFinder` object.

### Output Arguments

**nonempty — Whether queue is nonempty**

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.RequirementSetFinder` |  
`systemcomposer.rptgen.finder.RequirementSetResult` |  
`systemcomposer.rptgen.report.RequirementSet` | `find` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.RequirementSetFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next requirement set search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next `RequirementSet` search result.

### Input Arguments

**finder — Requirement set finder**

requirement set finder object

Requirement set finder, specified as a `systemcomposer.rptgen.finder.RequirementSetFinder` object.

### Output Arguments

**result — Requirement set result**

requirement set result object

Requirement set result, returned as a `systemcomposer.rptgen.finder.RequirementSetResult` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.RequirementSetFinder` |  
`systemcomposer.rptgen.finder.RequirementSetResult` |  
`systemcomposer.rptgen.report.RequirementSet` | `find` | `hasNext` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.RequirementSetResult`

**Package:** `systemcomposer.rptgen.finder`

Get requirements reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about requirements in a requirement set. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.RequirementSet` reporter class for more information on how to customize the reporter.

### Input Arguments

**result** — Requirement set result

requirement set result object

Requirement set result, specified as a `systemcomposer.rptgen.finder.RequirementSetResult` object.

### Output Arguments

**reporter** — Requirement set reporter

requirement set reporter object

Requirement set reporter, returned as a `systemcomposer.rptgen.report.RequirementSet` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.RequirementSetFinder` |  
`systemcomposer.rptgen.finder.RequirementSetResult` |  
`systemcomposer.rptgen.report.RequirementSet` | `find` | `hasNext` | `next` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# find

**Class:** `systemcomposer.rptgen.finder.StereotypeFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about stereotype

## Syntax

```
result = find(finder)
```

## Description

`result = find(finder)` finds information about a stereotype for the `Stereotype` search result.

## Input Arguments

**finder — Stereotype finder**

*stereotype finder object*

Stereotype finder, specified as a `systemcomposer.rptgen.finder.StereotypeFinder` object.

## Output Arguments

**result — Stereotype result**

*stereotype result object*

Stereotype result, returned as a `systemcomposer.rptgen.finder.StereotypeResult` object.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.StereotypeFinder` |  
`systemcomposer.rptgen.finder.StereotypeResult` |  
`systemcomposer.rptgen.report.Stereotype` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.StereotypeFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if stereotype search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the Stereotype search result queue is nonempty.

### Input Arguments

**finder — Stereotype finder**

*stereotype finder object*

Stereotype finder, specified as a `systemcomposer.rptgen.finder.StereotypeFinder` object.

### Output Arguments

**nonempty — Whether queue is nonempty**

*true or 1 | false or 0*

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

## Examples

### Generate Stereotype Finder Report

Use the `StereotypeFinder` and `StereotypeResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="StereotypeFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Stereotypes in %s Model',model_name)));
add(rpt,TableOfContents);
```

```
stereotypeFinder = StereotypeFinder("UAVComponent");
chapter = Chapter("Title","Stereotypes");
while hasNext(stereotypeFinder)
    stereotype = next(stereotypeFinder);
    sect = Section("Title",stereotype.Name);
    add(sect,stereotype);
    add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.StereotypeFinder |  
systemcomposer.rptgen.finder.StereotypeResult |  
systemcomposer.rptgen.report.Stereotype | find | next | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.StereotypeFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next stereotype search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next Stereotype search result.

### Input Arguments

**finder** — Stereotype finder

*stereotype finder object*

Stereotype finder, specified as a `systemcomposer.rptgen.finder.StereotypeFinder` object.

### Output Arguments

**result** — Stereotype result

*stereotype result object*

Stereotype result, returned as a `systemcomposer.rptgen.finder.StereotypeResult` object.

## Examples

### Generate Stereotype Finder Report

Use the `StereotypeFinder` and `StereotypeResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

scExampleSmallUAV
model_name = "scExampleSmallUAVModel";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="StereotypeFinderReport", ...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Stereotypes in %s Model',model_name)));
add(rpt,TableOfContents);

stereotypeFinder = StereotypeFinder("UAVComponent");
chapter = Chapter("Title","Stereotypes");
while hasNext(stereotypeFinder)
```

```
stereotype = next(stereotypeFinder);
sect = Section("Title",stereotype.Name);
add(sect,stereotype);
add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.StereotypeFinder |  
systemcomposer.rptgen.finder.StereotypeResult |  
systemcomposer.rptgen.report.Stereotype | find | hasNext | getReporter |  
createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.StereotypeResult`

**Package:** `systemcomposer.rptgen.finder`

Get stereotype reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about stereotypes in a profile. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.Stereotype` reporter class for more information on how to customize the reporter.

### Input Arguments

**result — Stereotype result**

*stereotype result object*

Stereotype result, specified as a `systemcomposer.rptgen.finder.StereotypeResult` object.

### Output Arguments

**reporter — Stereotype reporter**

*stereotype reporter object*

Stereotype reporter, returned as a `systemcomposer.rptgen.report.Stereotype` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.StereotypeFinder` |  
`systemcomposer.rptgen.finder.StereotypeResult` |  
`systemcomposer.rptgen.report.Stereotype` | `find` | `hasNext` | `next` | `createTemplate` |  
`customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# find

**Class:** `systemcomposer.rptgen.finder.ViewFinder`

**Package:** `systemcomposer.rptgen.finder`

Find information about view

## Syntax

```
result = find(finder)
```

## Description

`result = find(finder)` finds information about a view for the View search result.

## Input Arguments

**finder** – View finder

view finder object

View finder, specified as a `systemcomposer.rptgen.finder.ViewFinder` object.

## Output Arguments

**result** – View result

view result object

View result, returned as a `systemcomposer.rptgen.finder.ViewResult` object.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.ViewFinder` |  
`systemcomposer.rptgen.finder.ViewResult` | `systemcomposer.rptgen.report.View` |  
`hasNext` | `next` | `getReporter` | `createTemplate` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## hasNext

**Class:** `systemcomposer.rptgen.finder.ViewFinder`

**Package:** `systemcomposer.rptgen.finder`

Determine if view search result queue is nonempty

### Syntax

```
nonempty = hasNext(finder)
```

### Description

`nonempty = hasNext(finder)` determines whether the View search result queue is nonempty.

### Input Arguments

**finder** — View finder

view finder object

View finder, specified as a `systemcomposer.rptgen.finder.ViewFinder` object.

### Output Arguments

**nonempty** — Whether queue is nonempty

true or 1 | false or 0

Whether queue is nonempty, returned as a logical.

Data Types: `logical`

### Examples

#### Generate View Finder Report

Use the `ViewFinder` and `ViewResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

sckeylessEntrySystem
model_name = "KeylessEntryArchitecture";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ViewFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Views in %s Model',model_name)));
add(rpt,TableOfContents);

viewFinder = ViewFinder(model_name);
```



```
chapter = Chapter("Title", "Views");
while hasNext(viewFinder)
    view = next(viewFinder);
    sect = Section("Title", view.Name);
    add(sect, view);
    add(chapter, sect);
end

add(rpt, chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ViewFinder |  
systemcomposer.rptgen.finder.ViewResult | systemcomposer.rptgen.report.View |  
find | next | getReporter | createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## next

**Class:** `systemcomposer.rptgen.finder.ViewFinder`

**Package:** `systemcomposer.rptgen.finder`

Get next view search result

### Syntax

```
result = next(finder)
```

### Description

`result = next(finder)` gets the next View search result.

### Input Arguments

**finder** — View finder

view finder object

View finder, specified as a `systemcomposer.rptgen.finder.ViewFinder` object.

### Output Arguments

**result** — View result

view result object

View result, returned as a `systemcomposer.rptgen.finder.ViewResult` object.

## Examples

### Generate View Finder Report

Use the `ViewFinder` and `ViewResult` classes to generate a report.

```
import mlreportgen.report.*
import slreportgen.report.*
import systemcomposer.rptgen.finder.*

sckeylessEntrySystem
model_name = "KeylessEntryArchitecture";
model = systemcomposer.loadModel(model_name);
rpt = slreportgen.report.Report(output="ViewFinderReport",...
CompileModelBeforeReporting=false);
add(rpt,TitlePage("Title",sprintf('Views in %s Model',model_name)));
add(rpt,TableOfContents);

viewFinder = ViewFinder(model_name);

chapter = Chapter("Title", "Views");
```

```
while hasNext(viewFinder)
  view = next(viewFinder);
  sect = Section("Title",view.Name);
  add(sect,view);
  add(chapter,sect);
end

add(rpt,chapter);
close(rpt);
rptview(rpt);
```

## Version History

Introduced in R2022b

### See Also

systemcomposer.rptgen.finder.ViewFinder |  
systemcomposer.rptgen.finder.ViewResult | systemcomposer.rptgen.report.View |  
find | hasNext | getReporter | createTemplate | customizeReporter | getClassFolder

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

## getReporter

**Class:** `systemcomposer.rptgen.finder.ViewResult`

**Package:** `systemcomposer.rptgen.finder`

Get view reporter

### Syntax

```
reporter = getReporter(result)
```

### Description

`reporter = getReporter(result)` returns a reporter that is used to include information about views in a model. You can use this reporter to customize what information is included and how the information is formatted. See the `systemcomposer.rptgen.report.View` reporter class for more information on how to customize the reporter.

### Input Arguments

**result – View result**  
view result object

View result, specified as a `systemcomposer.rptgen.finder.ViewResult` object.

### Output Arguments

**reporter – View reporter**  
view reporter object

View reporter, returned as a `systemcomposer.rptgen.report.View` object.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.ViewFinder` |  
`systemcomposer.rptgen.finder.ViewResult` | `systemcomposer.rptgen.report.View` |  
`find` | `hasNext` | `next` | `createTemplate` | `customizeReporter` | `getClassFolder`

### Topics

“System Composer Report Generation for System Architectures”  
“System Composer Report Generation for Software Architectures”

# createTemplate

**Class:** systemcomposer.rptgen.report.AllocationList

**Package:** systemcomposer.rptgen.report

Create allocation list template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default allocation list template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom allocation list template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

systemcomposer.rptgen.finder.AllocationListFinder |  
systemcomposer.rptgen.finder.AllocationListResult |  
systemcomposer.rptgen.report.AllocationList | find | next | hasNext | getReporter |  
customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.AllocationList`

**Package:** `systemcomposer.rptgen.report`

Create custom allocation list reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates an allocation list class definition file that is a subclass of the `systemcomposer.rptgen.report.AllocationList` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default allocation list templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom allocation list class for your report.

## Input Arguments

**classpath — Location of custom allocation list class**

current working folder (default) | string | character array

Location of custom allocation list class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

**reporter — Allocation list reporter path**

string

Allocation list reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.AllocationListFinder` |  
`systemcomposer.rptgen.finder.AllocationListResult` |  
`systemcomposer.rptgen.report.AllocationList` | `find` | `next` | `hasNext` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.AllocationList`

**Package:** `systemcomposer.rptgen.report`

Allocation list class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the allocation list class definition file.

### Output Arguments

**path — Allocation list class definition file location**

character array

Allocation list class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.AllocationListFinder` |  
`systemcomposer.rptgen.finder.AllocationListResult` |  
`systemcomposer.rptgen.report.AllocationList` | `find` | `next` | `hasNext` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# createTemplate

**Class:** systemcomposer.rptgen.report.AllocationSet

**Package:** systemcomposer.rptgen.report

Create allocation set template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default allocation set template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom allocation set template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

systemcomposer.rptgen.finder.AllocationSetFinder |  
systemcomposer.rptgen.finder.AllocationSetResult |  
systemcomposer.rptgen.report.AllocationSet | find | hasNext | next | getReporter |  
customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.AllocationSet`

**Package:** `systemcomposer.rptgen.report`

Create custom allocation set reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates an allocation set class definition file that is a subclass of the `systemcomposer.rptgen.report.AllocationSet` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default allocation list templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom allocation set class for your report.

## Input Arguments

**classpath — Location of custom allocation set class**

current working folder (default) | string | character array

Location of custom allocation set class, specified as a string or character array. The `classpath` argument also supports specifying a folder with @ before the class name.

## Output Arguments

**reporter — Allocation set reporter path**

string

Allocation set reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.AllocationSetFinder` |  
`systemcomposer.rptgen.finder.AllocationSetResult` |  
`systemcomposer.rptgen.report.AllocationSet` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.AllocationSet`

**Package:** `systemcomposer.rptgen.report`

Allocation set class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the allocation set class definition file.

### Output Arguments

**path — Allocation set class definition file location**

character array

Allocation set class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.AllocationSetFinder` |  
`systemcomposer.rptgen.finder.AllocationSetResult` |  
`systemcomposer.rptgen.report.AllocationSet` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# createTemplate

**Class:** systemcomposer.rptgen.report.Component

**Package:** systemcomposer.rptgen.report

Create component template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default component template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom component template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

systemcomposer.rptgen.finder.ComponentFinder |  
systemcomposer.rptgen.finder.ComponentResult |  
systemcomposer.rptgen.report.Component | find | hasNext | next | getReporter |  
customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.Component`

**Package:** `systemcomposer.rptgen.report`

Create custom component reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a component class definition file that is a subclass of the `systemcomposer.rptgen.report.Component` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default component templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom component class for your report.

## Input Arguments

**classpath — Location of custom component class**

current working folder (default) | string | character array

Location of custom component class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

**reporter — Component reporter path**

string

Component reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.ComponentFinder` |  
`systemcomposer.rptgen.finder.ComponentResult` |  
`systemcomposer.rptgen.report.Component` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.Component`

**Package:** `systemcomposer.rptgen.report`

Component class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the component class definition file.

### Output Arguments

**path — Component class definition file location**

character array

Component class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.ComponentFinder` |  
`systemcomposer.rptgen.finder.ComponentResult` |  
`systemcomposer.rptgen.report.Component` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# createTemplate

**Class:** `systemcomposer.rptgen.report.Connector`

**Package:** `systemcomposer.rptgen.report`

Create connector template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default connector template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom connector template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.ConnectorFinder` |  
`systemcomposer.rptgen.finder.ConnectorResult` |  
`systemcomposer.rptgen.report.Connector` | `find` | `next` | `hasNext` | `getReporter` |  
`customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.Connector`

**Package:** `systemcomposer.rptgen.report`

Create custom connector reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a connector class definition file that is a subclass of the `systemcomposer.rptgen.report.Connector` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default connector templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom connector class for your report.

## Input Arguments

**classpath — Location of custom connector class**

current working folder (default) | string | character array

Location of custom connector class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

**reporter — Connector reporter path**

string

Connector reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.ConnectorFinder` |  
`systemcomposer.rptgen.finder.ConnectorResult` |  
`systemcomposer.rptgen.report.Connector` | `find` | `next` | `hasNext` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.Connector`

**Package:** `systemcomposer.rptgen.report`

Connector class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the connector class definition file.

### Output Arguments

**path — Connector class definition file location**

character array

Connector class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.ConnectorFinder` |  
`systemcomposer.rptgen.finder.ConnectorResult` |  
`systemcomposer.rptgen.report.Connector` | `find` | `next` | `hasNext` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# createTemplate

**Class:** `systemcomposer.rptgen.report.DependencyGraph`

**Package:** `systemcomposer.rptgen.report`

Create dependency graph template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default dependency graph template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom dependency graph template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdf.tx`.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.report.DependencyGraph` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.DependencyGraph`

**Package:** `systemcomposer.rptgen.report`

Create custom dependency graph reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a dependency graph class definition file that is a subclass of the `systemcomposer.rptgen.report.DependencyGraph` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default dependency graph templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom dependency graph class for your report.

## Input Arguments

**classpath — Location of custom dependency graph class**

current working folder (default) | string | character array

Location of custom dependency graph class, specified as a string or character array. The `classpath` argument also supports specifying a folder with @ before the class name.

## Output Arguments

**reporter — Dependency graph reporter path**

string

Dependency graph reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.report.DependencyGraph` | `createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.DependencyGraph`

**Package:** `systemcomposer.rptgen.report`

Dependency graph class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the dependency graph class definition file.

### Output Arguments

**path** — Dependency graph class definition file location

character array

Dependency graph class definition file location, returned as a character array.

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.report.DependencyGraph` | `createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# createTemplate

**Class:** systemcomposer.rptgen.report.Function

**Package:** systemcomposer.rptgen.report

Create function template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default function template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom function template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

systemcomposer.rptgen.finder.FunctionFinder |  
systemcomposer.rptgen.finder.FunctionResult |  
systemcomposer.rptgen.report.Function | find | hasNext | next | getReporter |  
customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.Function`

**Package:** `systemcomposer.rptgen.report`

Create custom function reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a function class definition file that is a subclass of the `systemcomposer.rptgen.report.Function` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default function templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom function class for your report.

## Input Arguments

**classpath — Location of custom function class**

current working folder (default) | string | character array

Location of custom function class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

**reporter — Function reporter path**

string

Function reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.FunctionFinder` |  
`systemcomposer.rptgen.finder.FunctionResult` |  
`systemcomposer.rptgen.report.Function` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** systemcomposer.rptgen.report.Function

**Package:** systemcomposer.rptgen.report

Function class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the function class definition file.

### Output Arguments

**path — Function class definition file location**

character array

Function class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

systemcomposer.rptgen.finder.FunctionFinder |  
systemcomposer.rptgen.finder.FunctionResult |  
systemcomposer.rptgen.report.Function | find | hasNext | next | getReporter |  
createTemplate | customizeReporter

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# createTemplate

**Class:** `systemcomposer.rptgen.report.Interface`

**Package:** `systemcomposer.rptgen.report`

Create interface template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default interface template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom interface template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.InterfaceFinder` |  
`systemcomposer.rptgen.finder.InterfaceResult` |  
`systemcomposer.rptgen.report.Interface` | `find` | `hasNext` | `next` | `getReporter` |  
`customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.Interface`

**Package:** `systemcomposer.rptgen.report`

Create custom interface reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a interface class definition file that is a subclass of the `systemcomposer.rptgen.report.Interface` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default interface templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom interface class for your report.

## Input Arguments

**classpath — Location of custom interface class**

current working folder (default) | string | character array

Location of custom interface class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

**reporter — Interface reporter path**

string

Interface reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.InterfaceFinder` |  
`systemcomposer.rptgen.finder.InterfaceResult` |  
`systemcomposer.rptgen.report.Interface` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.Interface`

**Package:** `systemcomposer.rptgen.report`

Interface class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the interface class definition file.

### Output Arguments

**path — Interface class definition file location**

character array

Interface class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.InterfaceFinder` |  
`systemcomposer.rptgen.finder.InterfaceResult` |  
`systemcomposer.rptgen.report.Interface` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# createTemplate

**Class:** systemcomposer.rptgen.report.Profile

**Package:** systemcomposer.rptgen.report

Create profile template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default profile template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom profile template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

systemcomposer.rptgen.finder.ProfileFinder |  
systemcomposer.rptgen.finder.ProfileResult |  
systemcomposer.rptgen.report.Profile | find | hasNext | next | getReporter |  
customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.Profile`

**Package:** `systemcomposer.rptgen.report`

Create custom profile reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a profile class definition file that is a subclass of the `systemcomposer.rptgen.report.Profile` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default profile templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom profile class for your report.

## Input Arguments

**classpath — Location of custom profile class**

current working folder (default) | string | character array

Location of custom profile class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

**reporter — Profile reporter path**

string

Profile reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.ProfileFinder` |  
`systemcomposer.rptgen.finder.ProfileResult` |  
`systemcomposer.rptgen.report.Profile` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.Profile`

**Package:** `systemcomposer.rptgen.report`

Profile class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the profile class definition file.

### Output Arguments

**path** — Profile class definition file location

character array

Profile class definition file location, returned as a character array.

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.finder.ProfileFinder` |  
`systemcomposer.rptgen.finder.ProfileResult` |  
`systemcomposer.rptgen.report.Profile` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# createTemplate

**Class:** systemcomposer.rptgen.report.RequirementLink

**Package:** systemcomposer.rptgen.report

Create requirement link template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default requirement link template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom requirement link template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

systemcomposer.rptgen.finder.RequirementLinkFinder |  
systemcomposer.rptgen.finder.RequirementLinkResult |  
systemcomposer.rptgen.report.RequirementLink | find | hasNext | next | getReporter |  
customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.RequirementLink`

**Package:** `systemcomposer.rptgen.report`

Create custom requirement link reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a requirement link class definition file that is a subclass of the `systemcomposer.rptgen.report.RequirementLink` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default requirement link templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom requirement link class for your report.

## Input Arguments

**classpath — Location of custom requirement link class**

current working folder (default) | string | character array

Location of custom requirement link class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

**reporter — Requirement link reporter path**

string

Requirement link reporter path, returned as a string specifying the path to the derived report class file.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.RequirementLinkFinder` |  
`systemcomposer.rptgen.finder.RequirementLinkResult` |  
`systemcomposer.rptgen.report.RequirementLink` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.RequirementLink`

**Package:** `systemcomposer.rptgen.report`

Requirement link class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the requirement link class definition file.

### Output Arguments

**path** — Requirement link class definition file location

character array

Requirement link class definition file location, returned as a character array.

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.finder.RequirementLinkFinder` |  
`systemcomposer.rptgen.finder.RequirementLinkResult` |  
`systemcomposer.rptgen.report.RequirementLink` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# createTemplate

**Class:** systemcomposer.rptgen.report.RequirementSet

**Package:** systemcomposer.rptgen.report

Create requirement set template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default requirement set template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom requirement set template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

systemcomposer.rptgen.finder.RequirementSetFinder |  
systemcomposer.rptgen.finder.RequirementSetResult |  
systemcomposer.rptgen.report.RequirementSet | find | hasNext | next | getReporter |  
customizeReporter | getClassFolder

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.RequirementSet`

**Package:** `systemcomposer.rptgen.report`

Create custom requirement set reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a requirement set class definition file that is a subclass of the `systemcomposer.rptgen.report.RequirementSet` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default requirement set templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom requirement set class for your report.

## Input Arguments

**classpath — Location of custom requirement set class**

current working folder (default) | string | character array

Location of custom requirement set class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

**reporter — Requirement set reporter path**

string

Requirement set reporter path, returned as a string specifying the path to the derived report class file.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.RequirementSetFinder` |  
`systemcomposer.rptgen.finder.RequirementSetResult` |  
`systemcomposer.rptgen.report.RequirementSet` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.RequirementSet`

**Package:** `systemcomposer.rptgen.report`

Requirement set class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the requirement set class definition file.

### Output Arguments

**path — Requirement set class definition file location**

character array

Requirement set class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.RequirementSetFinder` |  
`systemcomposer.rptgen.finder.RequirementSetResult` |  
`systemcomposer.rptgen.report.RequirementSet` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# createTemplate

**Class:** `systemcomposer.rptgen.report.Stereotype`

**Package:** `systemcomposer.rptgen.report`

Create stereotype template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default stereotype template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom stereotype template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdftx`.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.StereotypeFinder` |  
`systemcomposer.rptgen.finder.StereotypeResult` |  
`systemcomposer.rptgen.report.Stereotype` | `find` | `hasNext` | `next` | `getReporter` |  
`customizeReporter` | `getClassFolder`

**Topics**

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.Stereotype`

**Package:** `systemcomposer.rptgen.report`

Create custom stereotype reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a stereotype class definition file that is a subclass of the `systemcomposer.rptgen.report.Stereotype` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default stereotype templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom stereotype class for your report.

## Input Arguments

**classpath — Location of custom stereotype class**

current working folder (default) | string | character array

Location of custom stereotype class, specified as a string or character array. The `classpath` argument also supports specifying a folder with @ before the class name.

## Output Arguments

**reporter — Stereotype reporter path**

string

Stereotype reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.StereotypeFinder` |  
`systemcomposer.rptgen.finder.StereotypeResult` |  
`systemcomposer.rptgen.report.Stereotype` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.Stereotype`

**Package:** `systemcomposer.rptgen.report`

Stereotype class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the stereotype class definition file.

### Output Arguments

**path — Stereotype class definition file location**

character array

Stereotype class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.finder.StereotypeFinder` |  
`systemcomposer.rptgen.finder.StereotypeResult` |  
`systemcomposer.rptgen.report.Stereotype` | `find` | `hasNext` | `next` | `getReporter` |  
`createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# createTemplate

**Class:** `systemcomposer.rptgen.report.View`

**Package:** `systemcomposer.rptgen.report`

Create view template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default view template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom view template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdf.tx`.

## Version History

**Introduced in R2022b**

## See Also

`systemcomposer.rptgen.finder.ViewFinder` |  
`systemcomposer.rptgen.finder.ViewResult` | `systemcomposer.rptgen.report.View` |  
`find` | `hasNext` | `next` | `getReporter` | `customizeReporter` | `getClassFolder`

## Topics

"System Composer Report Generation for System Architectures"

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.View`

**Package:** `systemcomposer.rptgen.report`

Create custom view reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a view class definition file that is a subclass of the `systemcomposer.rptgen.report.View` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default view templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom view class for your report.

## Input Arguments

### **classpath** — Location of custom view class

current working folder (default) | string | character array

Location of custom view class, specified as a string or character array. The `classpath` argument also supports specifying a folder with `@` before the class name.

## Output Arguments

### **reporter** — View reporter path

string

View reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.finder.ViewFinder` |  
`systemcomposer.rptgen.finder.ViewResult` | `systemcomposer.rptgen.report.View` |  
`find` | `hasNext` | `next` | `getReporter` | `createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.View`

**Package:** `systemcomposer.rptgen.report`

View class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the view class definition file.

### Output Arguments

**path** — View class definition file location

character array

View class definition file location, returned as a character array.

## Version History

Introduced in R2022b

### See Also

`systemcomposer.rptgen.finder.ViewFinder` |  
`systemcomposer.rptgen.finder.ViewResult` | `systemcomposer.rptgen.report.View` |  
`find` | `hasNext` | `next` | `getReporter` | `createTemplate` | `customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# createTemplate

**Class:** `systemcomposer.rptgen.report.SequenceDiagram`

**Package:** `systemcomposer.rptgen.report`

Create sequence diagram template

## Syntax

```
template = createTemplate(templatePath,type)
```

## Description

`template = createTemplate(templatePath,type)` creates a copy of the default sequence diagram template specified by `type` at the location specified by `templatePath`. Use the copied template as a starting point to design a custom sequence diagram template for your report.

## Input Arguments

**templatePath — Path and file name of new template**

character vector | string scalar

Path and file name of the new template, specified as a character vector or string scalar.

**type — Type of template**

"html" | "html-file" | "docx" | "pdf"

Type of template, specified as "html", "html-file", "docx", or "pdf".

## Output Arguments

**template — Path and file name of template copy**

string scalar

Path and file name of the template copy, returned as a string scalar. The specified template type determines the file name extension of the template. For example, if the `type` argument is 'pdf', the file name extension is `.pdf.tx`.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.report.SequenceDiagram` | `customizeReporter` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

# customizeReporter

**Class:** `systemcomposer.rptgen.report.SequenceDiagram`

**Package:** `systemcomposer.rptgen.report`

Create custom sequence diagram reporter class

## Syntax

```
reporter = customizeReporter(classpath,type)
```

## Description

`reporter = customizeReporter(classpath,type)` creates a sequence diagram class definition file that is a subclass of the `systemcomposer.rptgen.report.SequenceDiagram` class. The file is created at the specified `classpath` location. The `customizeReporter` method also copies the default sequence diagram templates to the `<classpath>/resources/template` folder. Use the new class definition file as a starting point to design a custom sequence diagram class for your report.

## Input Arguments

**classpath — Location of custom sequence diagram class**

current working folder (default) | string | character array

Location of custom sequence diagram class, specified as a string or character array. The `classpath` argument also supports specifying a folder with @ before the class name.

## Output Arguments

**reporter — Sequence diagram reporter path**

string

Sequence diagram reporter path, returned as a string specifying the path to the derived report class file.

## Version History

Introduced in R2022b

## See Also

`systemcomposer.rptgen.report.SequenceDiagram` | `createTemplate` | `getClassFolder`

## Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”

## getClassFolder

**Class:** `systemcomposer.rptgen.report.SequenceDiagram`

**Package:** `systemcomposer.rptgen.report`

Sequence diagram class definition file location

### Syntax

```
path = getClassFolder
```

### Description

`path = getClassFolder` returns the path of the folder that contains the sequence diagram class definition file.

### Output Arguments

**path — Sequence diagram class definition file location**

character array

Sequence diagram class definition file location, returned as a character array.

## Version History

**Introduced in R2022b**

### See Also

`systemcomposer.rptgen.report.SequenceDiagram | createTemplate | customizeReporter`

### Topics

“System Composer Report Generation for System Architectures”

“System Composer Report Generation for Software Architectures”



# Tools and Apps

---

# Allocation Editor

Create and manage model-to-model allocations

## Description

Use the **Allocation Editor** in System Composer to establish traceable and directed relationships between architectural elements. Allocate components, ports, and connectors in a source model to architectural elements in a target model.

You can use allocations to establish relationships from software components to hardware components and to indicate deployment strategies. Allocate different instances of components, ports, and connectors and use allocations to perform various analyses, for example, resource-based allocation analysis.

The screenshot displays the Allocation Editor interface. On the left, the Allocation Set Browser shows a tree view with 'FunctionalAllocation' and 'PhysicalAllocation' folders, each containing 'Scenario 1'. The main area is a table for 'Scenario 1' with columns for hardware architectures: 'scMobileRobotHardwareArchitecture', 'Controller', 'Lidar Sensor', 'Target Machine', 'RGB Camera', and 'Mobile Robot Case'. The table rows represent software components: 'scMobileRobotLogicalArchitecture\_SS', 'Robot Body', 'Sensor Processing', 'Scan Matching Algorithm', 'Sensor Fusion', 'Alignment Algorithm', and 'Trajectory Generator'. The 'Scan Matching Algorithm' row is highlighted, and its properties are shown in the right pane. The Allocation Properties pane includes fields for Name, Allocated (checkbox), Source (Component), Main (Name: Scan Matching Algorithm), Parameters (No parameters defined), Target (Component), Main (Name: Mobile Robot Case), MechanicalComponent (Name, Mass: 3 kg, Life: 6000 hours, UsagePerDay: 24 hours, UsagePerYear: 365 days, ExceedExpectedMaintenance: false).

## Open the Allocation Editor

- System Composer toolstrip: Navigate to **Modeling > Allocation Editor**.
- MATLAB Command Window: Enter `systemcomposer.allocation.editor`.

## Examples

- “Create and Manage Allocations Interactively”
- “Create and Manage Allocations Programmatically”
- “Allocate Architectures in Tire Pressure Monitoring System”
- “Systems Engineering Approach for SoC Applications”

## Parameters

**New Allocation Set** — Create new allocation set

button

Create a new allocation set saved as an MLDATX file. Within the allocation set, add allocation scenarios.

**Add Scenario** — Add allocation scenario

button

Add an allocation scenario in the selected allocation set. Within the allocation scenario, allocate elements between two architecture models.

**Synchronize** — Synchronize changes of models in allocation set

button

This button synchronizes any changes that have been made in the source or target models of the allocation set. To synchronize changes programmatically, see `synchronizeChanges`.

**Filters** — Row filter and column filter

button

Choose a row filter and a column filter. Filter all allocation scenarios by a combination of the following options:

- Port
- Connector
- Component
- Allocated
- Un-Allocated

You can also filter by one or more stereotypes.

Select **Clear All Filters** to clear every filter, **Clear Row Filters** to clear row filters, or **Clear Column Filters** to clear column filters.

## Programmatic Use

`systemcomposer.allocation.editor` opens the **Allocation Editor** from the MATLAB Command Window.

## More About

### Allocation

An allocation establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model.

Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements.

### Allocation Scenario

An allocation scenario contains a set of allocations between a source and a target model.

Allocate between model elements in an allocation scenario. The default allocation scenario is called Scenario 1.

### Allocation Set

An allocation set consists of one or more allocation scenarios that describe various allocations between a source and a target model.

Create an allocation set with allocation scenarios in the **Allocation Editor**. Allocation sets are saved as MLDATX files.

## Version History

**Introduced in R2020b**

### See Also

`systemcomposer.allocation.AllocationScenario` |  
`systemcomposer.allocation.AllocationSet` | `editor` | `getScenario` | `allocate` |  
`synchronizeChanges`

### Topics

“Create and Manage Allocations Interactively”

“Create and Manage Allocations Programmatically”

“Allocate Architectures in Tire Pressure Monitoring System”

“Systems Engineering Approach for SoC Applications”

# Analysis Viewer

View and edit analysis instance model and analyze using analysis function

## Description

The **Analysis Viewer** shows an instantiated architecture.

The **Analysis Viewer** shows all elements in the first column. The other columns show properties for all stereotypes chosen for the current instance. If a property is not part of a stereotype applied to an element, that field is greyed out. You can use the **Filter** button to hide properties for certain stereotypes. When you select an element, **Instance Properties** shows the stereotypes and property values of the element. You can save an instance in a MAT-file and open it again in the **Analysis Viewer**.

Instances	totalPrice	unitPrice	weight	length	weight	ID
ex_RobotArch_props	686	5	0			
Motion	165	150	7			
Encoder	0	5				
MotionCommand	0	5				
SensorData	0	5				
Sensors	156	78	0			
Adapter	5	5	0			
Adapter.In->Adapter.Out	0	5		0	0	
DataProcessing	66	56	5			
OutBus	0	5				
RawData	0	5				
GPS	10	5	47			
GPSData	0	5				
GyroData	15	5	21			
InBus	0	5				
MotionData	0	5				
Adapter.Out->DataProcessing.RawData	0	5		2	12	
DataProcessing.OutBus->Sensors.SensorData	0	5		1	12	
GPS.GPSData->Adapter.In	0	5		3	12	
GyroData.MotionData->Adapter.InBus	0	5		3	12	
Sensors.Encoder->GyroData.InBus	0	5		1	12	
Encoder	0	5				
SensorData	0	5				
Trajectory Planning	240	45	0			
MotionController	75	60	4			
SensorData	0	5				
TargetPosition	0	5				
command	0	5				
SafetyRules	95	80	4			
OutBus	0	5				
SensorData	0	5				
command	0	5				

## Open the Analysis Viewer

- System Composer toolstrip: Navigate to **Modeling > Analysis Model > Analysis Viewer**.
- In the **Instantiate Architecture Model** tool, select **Instantiate**.

## Examples

- “Analyze Architecture”

- “Analysis Function Constructs”
- “Simple Roll-Up Analysis Using Robot System with Properties”
- “Define Stereotypes and Perform Analysis”
- “Calculate Endurance Using Quadcopter Architectural Design”
- “Design Insulin Infusion Pump Using Model-Based Systems Engineering”

## Parameters

**New** — Create new instance model  
button

Create a new instance model using the **Instantiate Architecture Model** tool.

**Open** — Open instance model  
button

Open a saved MAT file of an existing instance model.

**Save** — Save instance model  
button

Save the current instance model as a MAT file.

**Delete** — Delete instance model  
button

Delete the current instance model.

**Analyze** — Analyze architecture instance  
button

Analyze the architecture instance using an analysis function.

**Arguments** — Analysis arguments  
comma-separated values

Comma-separated values of optional arguments to the analysis function.

**Iteration Order** — Iteration type  
Preorder | Postorder | TopDown | BottomUp

Iteration type to specify how to process instances while using the analysis function. Select one of these options from the list:

- **Pre-order** — Start from the top level, move to a child component, and process the subcomponents of that component recursively before moving to a sibling component.
- **Top-Down** — Like pre-order, but process all sibling components before moving to their subcomponents.
- **Post-order** — Start from components with no subcomponents, process each sibling, and then move to parent.

- **Bottom-up** — Like post-order, but process all subcomponents at the same depth before moving to their parents.

**Update** — Push changes from instance to model  
button

Push the changes from the architecture instance to the architecture model.

**Refresh** — Pull changes to instance from model  
button

Pull changes to the architecture instance from the architecture model.

**Continuous** — Whether continuous analysis is enabled when values change  
off (default) | on

Select this check box to enable continuous analysis when values change.

**Automatic** — Whether instance automatically refreshes when composition changes  
off (default) | on

Select this check box to automatically refresh the instance when the composition changes.

**Overwrite** — Whether to overwrite entire instance model from composition model  
off (default) | on

Select this check box to overwrite the entire instance model from the composition model.

## Programmatic Use

`systemcomposer.analysis.loadInstance` loads a saved architecture instance object from a saved MAT-file that can be later opened in the **Analysis Viewer**.

## More About

### Analysis

Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.

Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.

### Analysis Function

An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.

Use an analysis function to calculate the result of an analysis.

## **Instance Model**

An instance model is a collection of instances.

You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.

### **Instance**

An instance is an occurrence of an architecture model element at a given point in time.

An instance freezes the active variant or model reference of the component in the instance model.

## **Version History**

**Introduced in R2019a**

### **See Also**

`instantiate` | `iterate` | `lookup` | `save` | `update` | `refresh` |  
`systemcomposer.analysis.loadInstance` | `systemcomposer.analysis.deleteInstance` |  
`getValue` | `setValue` | `hasValue` | `isArchitecture` | `isComponent` | `isConnector` | `isPort`

### **Topics**

“Analyze Architecture”

“Analysis Function Constructs”

“Simple Roll-Up Analysis Using Robot System with Properties”

“Define Stereotypes and Perform Analysis”

“Calculate Endurance Using Quadcopter Architectural Design”

“Design Insulin Infusion Pump Using Model-Based Systems Engineering”



# Architecture Views Gallery

Create and manage architecture views and sequence diagrams

## Description

The **Architecture Views Gallery** allows you to create filtered and freeform architecture views and author sequence diagrams.

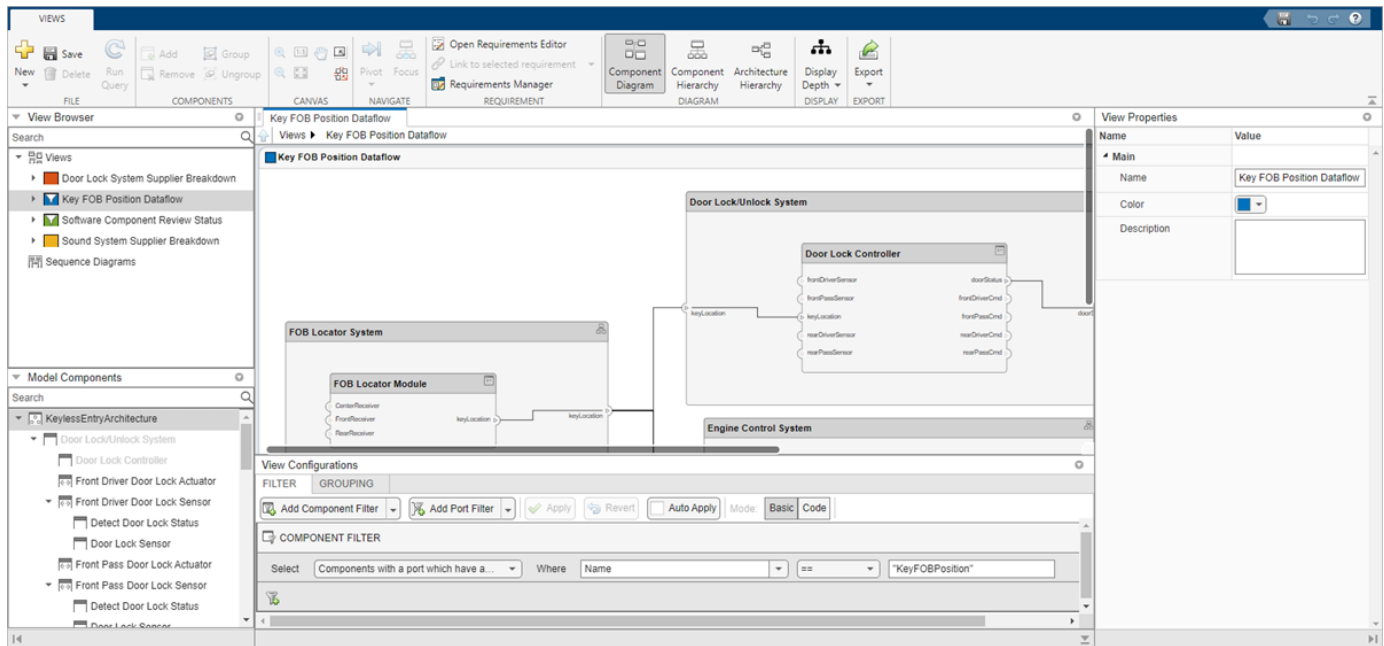
Use the **View Configurations** options to specify component and port filters for views, and to specify grouping criteria. Click and drag components from the **Model Components** browser to specify the contents of a freeform view. Select views from the **View Browser** and use the **Component Properties** options to specify a name, color, and description for a view.

Switch between these types of view diagrams:

- **Component Diagram** — Display components, ports, and connectors based on how the model is structured.
- **Component Hierarchy** — Display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.
- **Architecture Hierarchy** — Display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.
- **Class Diagram** — Display unique architecture types of the software components optionally with software methods and properties, only available for software architecture models.

You can also link and edit requirements for views through the **Architecture Views Gallery**.

To create a new sequence diagram, click **New > Sequence Diagram**. Select existing sequence diagrams from the **View Browser** and use the **Sequence Diagram Properties** options to specify a name for the sequence diagram. To add a lifeline, click and drag from the **Model Components** browser. Alternatively, select **Component > Add Lifeline** from the menu and click the down arrow to select a component to be represented by the lifeline. Click and drag from the vertical dotted lines coming down from one lifeline to another to author a message that represents a connection between two ports. To confirm the consistency of the sequence diagram, click **Check Consistency**. Then, either push changes to the architecture by clicking **Create in Architecture**, or pull changes in from the architecture to the sequence diagram by clicking **Repair**.



## Open the Architecture Views Gallery

- System Composer toolstrip: Navigate to **Modeling > Architecture Views**.
- System Composer toolstrip: Navigate to **Modeling > Sequence Diagram**.
- MATLAB Command Window: Enter `openViews` with a `systemcomposer.arch.Model` object as the input argument.

## Examples

- “Modeling System Architecture of Keyless Entry System”
- “Create Architectural Views Programmatically”
- “Create Architecture Views Interactively”
- “Display Component Hierarchy and Architecture Hierarchy Using Views”
- “Class Diagram View of Software Architectures”
- “Describe System Behavior Using Sequence Diagrams”
- “Simulate Sequence Diagrams for Traffic Light Example”

## Parameters

**New** — Create new view or sequence diagram button

Create a new view by default by clicking **New**, or click the drop-down arrow to choose **New > View**. Create a new sequence diagram by selecting **New > Sequence Diagram**.

**Save** — Save views, sequence diagrams, and model  
button

Save all views, sequence diagrams, and the architecture model.

**Delete** — Delete currently selected diagram  
button

Delete the currently selected view or sequence diagram.

**Run Query** — Refresh currently selected view  
button

Refresh the currently selected view with changes in the composition and rerun the corresponding filter, if it exists.

**Add** — Add selected component to view  
button

Add the selected component in the **Model Components** browser to the current view diagram.

If the view is a filtered view, a prompt appears to convert the filtered view to a freeform view.

**Remove** — Remove selected component from view  
button

Remove a selected component in a view from the current view diagram.

If the view is a filtered view, a prompt appears to convert the filtered view to a freeform view.

**Group** — Group selected components in view  
button

Group the selected components in a view.

**Ungroup** — Ungroup selected components in view  
button

Ungroup the selected components in a view.

**Pivot** — Pivot to other diagrams in which selected component or lifeline appears  
button

Pivot to other diagrams in which selected component or lifeline appears. Use the drop-down list to select the view diagram or sequence diagram to which to pivot. For more information, see “Pivot Between Lifelines and Components in Views Gallery”.

**Focus** — Focus on selected component  
button

Focus on the selected component to make it the new root of the diagram in the view.

**Display Depth** — Modify number of levels of hierarchy to display  
**Deep** (default) | **Shallow**

Modify the number of levels of hierarchy to display. **Deep** includes more levels and **Shallow** includes fewer levels.

**Export** — Export to image  
button

Export the currently selected diagram as an image. View diagrams can be saved as PDF files. Sequence diagrams can be saved as PDF files or image files.

**Add Lifeline** — Insert new lifeline into sequence diagram  
button

Create a new lifeline after the selected lifeline by default by clicking **Add Lifeline**, or click the drop-down arrow to choose **Add Lifeline > Insert After**. Create a new lifeline before the selected lifeline by selecting **Add Lifeline > Insert Before**. Create a new lifeline nested under the selected lifeline by selecting **Add Lifeline > Add Child**.

**Add Operand** — Insert new operand into sequence diagram  
button

Create a new operand after the selected operand by default by clicking **Add Operand**, or click the drop-down arrow to choose **Add Operand > Insert After**. Create a new operand before the selected operand by selecting **Add Operand > Insert Before**.

**Check Consistency** — Check whether elements in sequence diagram are consistent with architecture model  
button

Check that all the elements in the current sequence diagram are consistent with the architecture model. If any of the elements in the sequence diagram are inconsistent, clicking **Check Consistency** highlights those elements in yellow.

**Architecture Element** — Specify different associated element in architecture model for selected elements in sequence diagram  
component | port

Specify a different associated element in the architecture model for the selected elements in the sequence diagram.

**Create in Architecture** — Create elements in architecture model  
button

Create elements in the architecture model for each of the selected inconsistent elements in the sequence diagram.

**Repair** — Update selected elements so sequence diagram is consistent with architecture model  
button

Update the selected inconsistent elements in the sequence diagram so the sequence diagram is consistent with the architecture model.

**Run** — Run simulation  
button

Run model simulation and verify that the model simulation results match the interactions within the sequence diagrams.

**Pause** — Pause simulation  
button

Pause model simulation and sequence diagram simulation.

**Stop** — Stop simulation  
button

Stop model simulation and sequence diagram simulation.

**Continue** — Continue simulation  
button

Continue model simulation until the end and verify that the model simulation results match the interactions within the sequence diagrams.

**Next Message** — Continue until next message is hit  
button

Continue until next message is hit and verify that the model simulation results match the interactions within the sequence diagrams.

**Clear Results** — Clear simulation results  
button

Clear simulation results and remove green check marks or red warning marks on the sequence diagram.

## Programmatic Use

`openViews(model)` opens the **Architecture Views Gallery** from the MATLAB Command Window.

## More About

### View

A view shows a customizable subset of elements in a model. Views can be filtered based on stereotypes or names of components, ports, and interfaces, along with the name, type, or units of an interface element. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.

You can use different types of views to represent the system. Switch between a component diagram, component hierarchy, or architecture hierarchy. For software architectures, you can switch to a class diagram view.

### Element Group

An element group is a grouping of components in a view.

Use element groups to programmatically populate a view.

## Query

A query is a specification that describes certain constraints or criteria to be satisfied by model elements.

Use queries to search elements with constraint criteria and to filter views.

## Component Diagram

A component diagram represents a view with components, ports, and connectors based on how the model is structured.

Component diagrams allow you to programmatically or manually add and remove components from the view.

## Hierarchy Diagram

You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.

There are two types of hierarchy diagrams:

- *Component hierarchy diagrams* display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.
- *Architecture hierarchy diagrams* display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.

## Class Diagram

A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.

Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.

## Sequence Diagram

A sequence diagram represents the expected interaction between structural elements of an architecture as a sequence of message exchanges.

Use sequence diagrams to describe how the parts of a system interact.

### Lifeline

A lifeline is represented by a head and a timeline that proceeds down a vertical dotted line.

The head of a lifeline represents a component in an architecture model.

### Message

A message sends information from one lifeline to another. Messages are specified with a message label.

A message label has a trigger and a constraint. A trigger determines whether the message occurs. A constraint determines whether the message is valid.

### **Annotation**

An annotation describes the elements of a sequence diagram.

Use annotations to provide detailed explanations of elements or workflows captured by sequence diagrams.

### **Fragment**

A fragment indicates how a group of messages within it execute or interact.

A fragment is used to model complex sequences, such as alternatives, in a sequence diagram.

### **Operand**

An operand is a region in a fragment. Fragments have one or more operands depending on the kind of fragment. Operands can contain messages and additional fragments.

Each operand can include a constraint to specify whether the messages inside the operand execute. You can express the precondition of an operand as a MATLAB Boolean expression using the input signal of any lifeline.

## **Version History**

**Introduced in R2019b**

### **See Also**

#### **Functions**

`find` | `lookup` | `createView` | `getView` | `openViews` | `deleteView`

#### **Objects**

`systemcomposer.query.Constraint` | `systemcomposer.view.View` | `systemcomposer.view.ElementGroup`

#### **Topics**

“Modeling System Architecture of Keyless Entry System”

“Create Architectural Views Programmatically”

“Create Architecture Views Interactively”

“Display Component Hierarchy and Architecture Hierarchy Using Views”

“Class Diagram View of Software Architectures”

“Describe System Behavior Using Sequence Diagrams”

“Simulate Sequence Diagrams for Traffic Light Example”

## Comparison Tool

View differences between two architecture models




### Description

The **Comparison Tool** in System Composer shows differences between two architecture models.

The tool shows differences for these types of architectural data:

- Model structural differences (components, ports, and connectors)
- Different types of supported components and ports
- Interfaces on model data dictionaries
- Owned port interfaces
- Applied stereotypes and property value changes on model elements
- Architecture views
- Parameters
- Simulink properties

Rows in the comparison report are highlighted according to the type of difference:

- Insertion  — Added elements to the right side that did not exist on the left side
- Deletion  — Removed elements that did exist on the left side but not on the right side
- Modification  — Changes to existing elements that exist on both the left and right sides

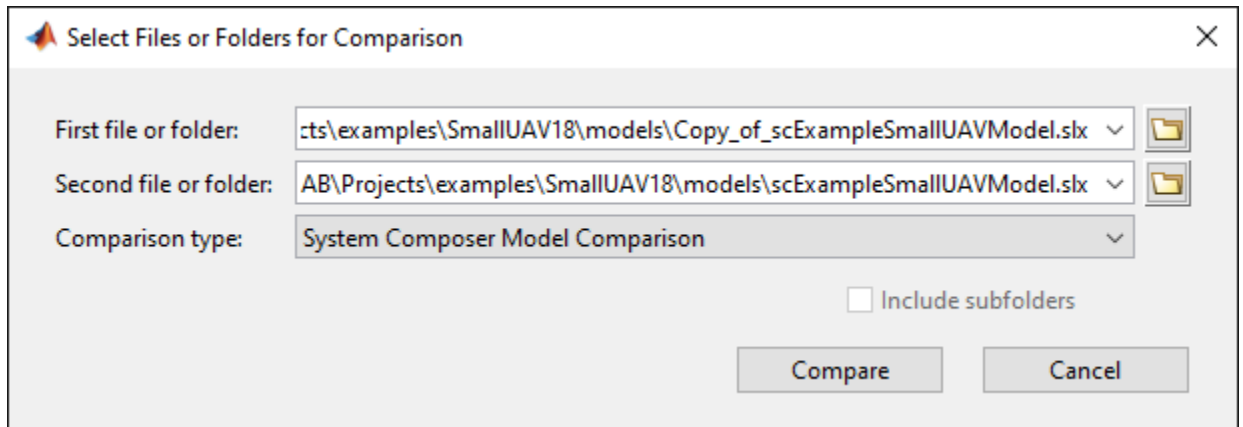


Architecture Property	Value	Architecture Property	Value
Name	scMobileRobotHardwareArchitecture	Name	scMobileRobotHardwareArchitectureEdit...
Simulink Property	Value	Simulink Property	Value
OrderedModelArguments	on	OrderedModelArguments	off

Insertion  
 Deletion  
 Modification

## Open the Comparison Tool

- Open the **Comparison Tool** from the System Composer toolstrip.
  - 1 Navigate to **Modeling > Compare**.
  - 2 In the Select Files or Folders for Comparison dialog box, select the second file against which to compare.
  - 3 Set the comparison type as System Composer Model Comparison.
  - 4 Click **Compare**.



- Open the **Comparison Tool** from the MATLAB® Current Folder browser by selecting one architecture model.
  - 1 In the MATLAB® Current Folder browser, right-click an architecture model.
  - 2 Select **Compare Against** and then **Choose**.
  - 3 In the Select Files or Folders for Comparison dialog box, select the second file against which to compare.
  - 4 Set the comparison type as **System Composer Model Comparison**.
  - 5 Click **Compare**.
- Open the **Comparison Tool** from the MATLAB® Current Folder browser by selecting two architecture models.
  - 1 In the MATLAB® Current Folder browser, select two architecture models.
  - 2 Right-click and select **Compare Selected Files/Folders**.

## Examples

- “Compare Model Differences Using System Composer Comparison Tool”
- “Compose Architectures Visually”
- “Define Port Interfaces Between Components”
- “Define Profiles and Stereotypes”
- “Create Architecture Views Interactively”
- “Implement Component Behavior Using Simulink”

## Parameters

**Swap Sides** — Switch left and right comparison models  
button

Swap sides of the two models being compared on the comparison report.

**Refresh** — Pull changes from architecture models to comparison report  
button

When the architecture models are out of sync, pull in the changes to the comparison report. You must save both architecture models first before clicking **Refresh**.

**Highlight Now** — Highlight currently selected report node  
button

When **Always Highlight** is turned off, you can click **Highlight Now** to highlight the currently selected comparison report node in the architecture models.

**Always Highlight** — Whether to always highlight differences in models  
on (default) | off

By default, the two models being compared display to the right of the comparison report, with the model corresponding to the left side of the report on top and the model corresponding to the right side appearing below. Turn **Always Highlight** off to use the **Highlight Now** button and control highlighting in the models.

**Hide Graphical Changes** — Whether to hide graphical changes from comparison models  
on (default) | off

Access this check box from the **Filter** menu. When selected, graphical changes such as component positioning and resizing are ignored from the comparison report.

## Programmatic Use

`visdiff("scMobileRobot.slx","scMobileRobotEdited.slx")` opens the **Comparison Tool** from the MATLAB Command Window.

## Version History

Introduced in R2022a

### See Also

`visdiff`

### Topics

- “Compare Model Differences Using System Composer Comparison Tool”
- “Compose Architectures Visually”
- “Define Port Interfaces Between Components”
- “Define Profiles and Stereotypes”
- “Create Architecture Views Interactively”
- “Implement Component Behavior Using Simulink”

# Functions Editor

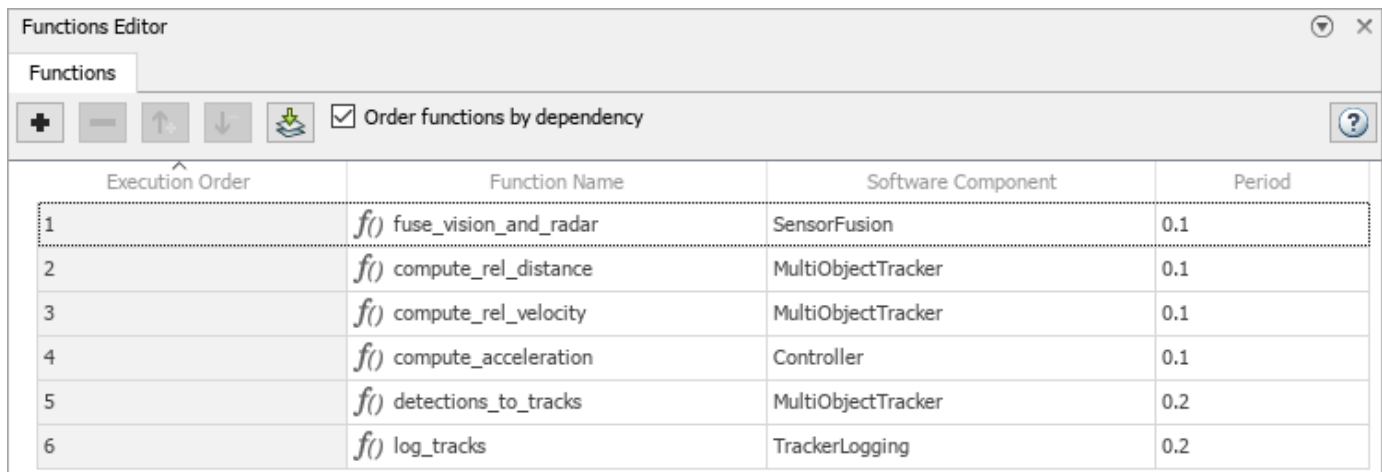
Visualize and author component functions in software architectures

## Description

The **Functions Editor** allows you to author functions in the architecture level for inline components. You can then implement Simulink behaviors for your authored functions. For reference components, the functions are automatically created from the referenced behavior Simulink models.

Use the **Functions Editor** to:

- Author and visualize functions.
  - Add and delete functions.
  - Change the execution order of the functions.
  - Change the name of a function.
  - Change the period of a function.
- Implement behaviors for functions.
- Add custom properties to functions using stereotypes.



Execution Order	Function Name	Software Component	Period
1	<i>f()</i> fuse_vision_and_radar	SensorFusion	0.1
2	<i>f()</i> compute_rel_distance	MultiObjectTracker	0.1
3	<i>f()</i> compute_rel_velocity	MultiObjectTracker	0.1
4	<i>f()</i> compute_acceleration	Controller	0.1
5	<i>f()</i> detections_to_tracks	MultiObjectTracker	0.2
6	<i>f()</i> log_tracks	TrackerLogging	0.2

## Open the Functions Editor

- System Composer toolstrip: Navigate to **Modeling > Functions Editor**.


## Examples

- “Authoring Functions for Software Components of an Adaptive Cruise Control”
- “Author and Extend Functions for Software Architectures”
- “Define Profiles and Stereotypes”

- “Use Property Inspector in System Composer”

## Parameters


**Add function** — Add function to software component  
button

Add a function to a software component by clicking .

**Remove function** — Remove function from software component  
button


Remove a function from a software component by clicking .

**Increase execution order** — Increase execution order of function  
button

Increase the execution order of a function by clicking .


This option is only available if **Order functions by dependency** is unchecked.

**Decrease execution order** — Decrease execution order of function  
button

Decrease the execution order of a function by clicking .

This option is only available if **Order functions by dependency** is unchecked.

**Update diagram** — Update diagram to refresh functions  
button

Update the software architecture diagram to refresh the functions in the **Functions Editor** by clicking .

**Order functions by dependency** — Whether to order functions by dependency  
off (default) | on

Select this check box to order functions in the **Functions Editor** by dependency.

You can order functions automatically based on their data dependencies. This functionality is available for functions from behavior models. To enable automatic sorting, select the **Order functions by dependency** check box or enable `OrderFunctionsByDependency` on the architecture model.

## Programmatic Use

Use the `addFunction` function to author functions. Use the `createSimulinkBehavior` function to create new Simulink rate-based or export-function behaviors and link the software component to the new model.

## More About

### Software Architecture

A software architecture is a specialization of an architecture for software-based systems, including the description of software compositions, component functions, and their scheduling.

Use software architectures in System Composer to author software architecture models composed of software components, ports, and interfaces. Design your software architecture model, define the execution order of your component functions, simulate your design in the architecture level, and generate code.

### Software Component

A software component is a specialization of a component for software entities, including its functions (entry points) and interfaces.

Implement a Simulink export-function, rate-based, or JMAAB model as a software component, simulate the software architecture model, and generate code.

### Software Composition

A software composition is a diagram of software components and connectors that represents a composite software entity, such as a module or application.

Encapsulate functionality by aggregating or nesting multiple software components or compositions.

### Function

A function is an entry point that can be defined in a software component.

You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the **Functions Editor**.

### Service Interface

A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.

Once you have defined a service interface in the **Interface Editor**, you can assign it to client and server ports using the **Property Inspector**. You can also use the **Property Inspector** to assign stereotypes to service interfaces.

### Function Element

A function element describes the attributes of a function in a client-server interface.

Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:

- Synchronous execution — When the client calls the server, the function runs immediately and returns the output arguments to the client.

- Asynchronous execution — When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the **Functions Editor** and **Schedule Editor** and returns the output arguments to the client.

### Function Argument

A function argument describes the attributes of an input or output argument in a function element.

You can set the properties of a function argument in the **Interface Editor** just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.

## Version History

Introduced in R2021b

### See Also

`systemcomposer.arch.Function` | `systemcomposer.interface.ServiceInterface` | `systemcomposer.interface.FunctionElement` | `systemcomposer.interface.FunctionArgument` | `addFunction` | `decreaseExecutionOrder` | `increaseExecutionOrder` | `addServiceInterface` | `setFunctionPrototype` | `getFunctionArgument`

### Topics

“Authoring Functions for Software Components of an Adaptive Cruise Control”

“Author and Extend Functions for Software Architectures”

“Define Profiles and Stereotypes”

“Use Property Inspector in System Composer”

## Instantiate Architecture Model

Create an instance of the architecture model that you can use for analysis

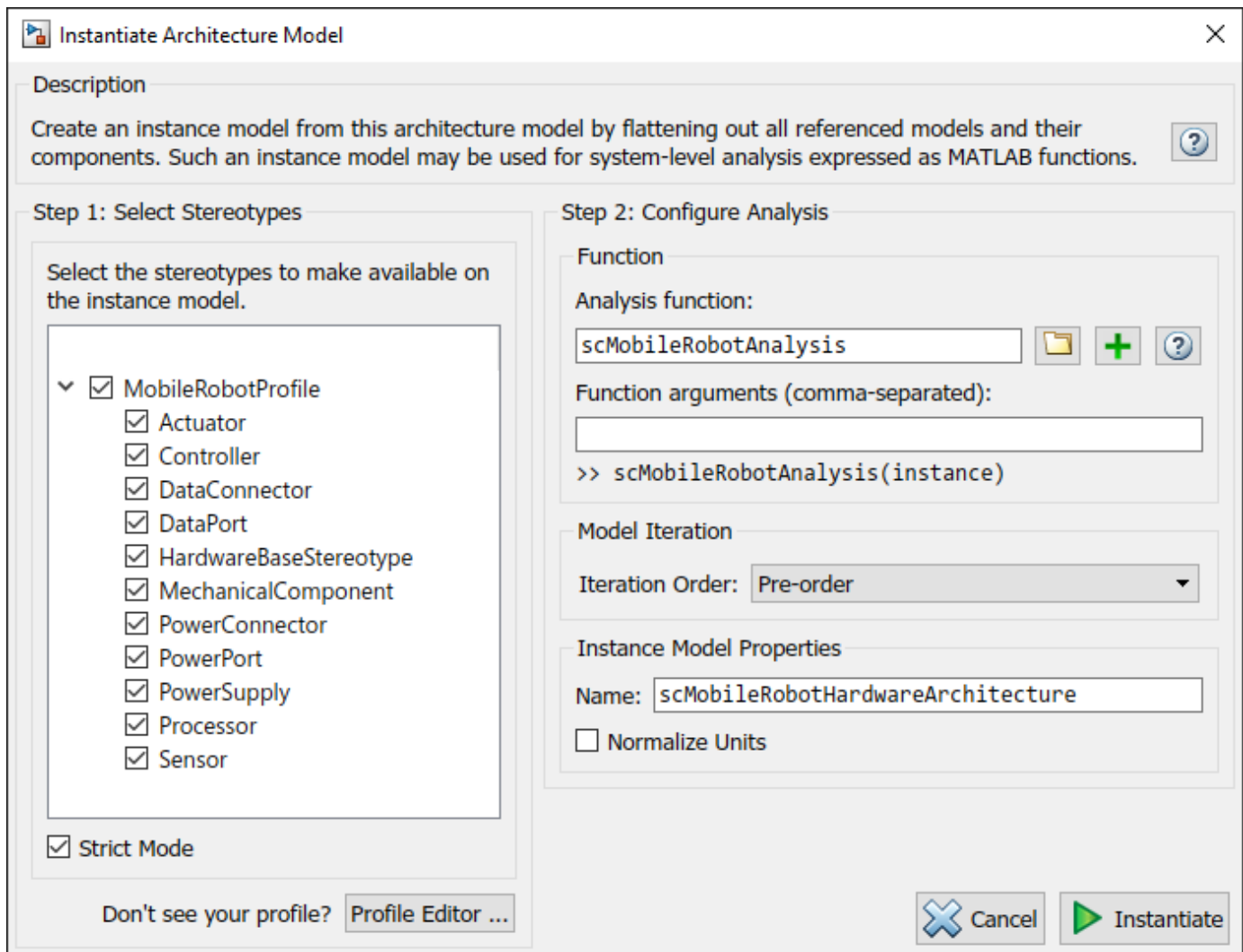
### Description

**Instantiate Architecture Model** creates an instance of an architecture model for analysis.

The **Select Stereotypes** tree lists the stereotypes of all profiles that have been loaded in the current session and allows you to select those whose properties should be available in the instance model. You can browse for an analysis function, create a new analysis function, or skip analysis at this point. If the analysis function requires inputs other than elements in the model, such as an exchange rate to compute cost, enter it in **Function arguments**. Select a mode for iterating through model elements, for example, **Bottom-up** to move from the leaves of the tree to the root. **Strict Mode** ensures elements in the instantiated model get properties only if the corresponding element in the composition model has the stereotype applied.

Click **Instantiate** to open the **Analysis Viewer**.





## Open the Instantiate Architecture Model

- System Composer toolstrip: Navigate to **Modeling > Analysis Model**.

## Examples

- “Analyze Architecture”
- “Analysis Function Constructs”
- “Simple Roll-Up Analysis Using Robot System with Properties”
- “Define Stereotypes and Perform Analysis”
- “Calculate Endurance Using Quadcopter Architectural Design”
- “Design Insulin Infusion Pump Using Model-Based Systems Engineering”

## Parameters

### **Analysis Function** — Analysis function

M-file

Analysis function, specified as the MATLAB function handle to be executed when analysis is run. For more information, see “Analysis Function Constructs”.

### **Function arguments** — Analysis arguments

comma-separated values

Comma-separated values of optional arguments to the analysis function.

### **Iteration Order** — Iteration type

Pre-order | Post-order | Top-Down | Bottom-up

Iteration type to specify how to process instances while using the analysis function. Select one of these options from the list:

- **Pre-order** — Start from the top level, move to a child component, and process the subcomponents of that component recursively before moving to a sibling component.
- **Top-Down** — Like pre-order, but process all sibling components before moving to their subcomponents.
- **Post-order** — Start from components with no subcomponents, process each sibling, and then move to parent.
- **Bottom-up** — Like post-order, but process all subcomponents at the same depth before moving to their parents.

### **Normalize Units** — Whether to normalize value based on units

off (default) | on

Whether to normalize value based on units, if any, specified in property definition upon instantiation.

### **Strict Mode** — Condition for instances getting properties

off (default) | on

Condition for instances getting properties only if the corresponding element in the composition model has the stereotype applied.

## Programmatic Use

Use the `instantiate` function or the `iterate` function for programmatic analyses.

## More About

### **Analysis**

Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.

Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.

### **Analysis Function**

An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.

Use an analysis function to calculate the result of an analysis.

### **Instance Model**

An instance model is a collection of instances.

You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.

### **Instance**

An instance is an occurrence of an architecture model element at a given point in time.

An instance freezes the active variant or model reference of the component in the instance model.

## **Version History**

**Introduced in R2019a**

### **See Also**

`instantiate` | `iterate` | `lookup` | `save` | `update` | `refresh` |  
`systemcomposer.analysis.loadInstance` | `systemcomposer.analysis.deleteInstance` |  
`getValue` | `setValue` | `hasValue` | `isArchitecture` | `isComponent` | `isConnector` | `isPort`

### **Topics**

“Analyze Architecture”

“Analysis Function Constructs”

“Simple Roll-Up Analysis Using Robot System with Properties”

“Define Stereotypes and Perform Analysis”

“Calculate Endurance Using Quadcopter Architectural Design”

“Design Insulin Infusion Pump Using Model-Based Systems Engineering”

## Interface Editor

Create and author interfaces in local and shared interface data dictionaries

### Description

The **Interface Editor** allows you to define interfaces in System Composer that might contain attributes. In System Composer architecture models, interfaces are necessary to specify information that flows through ports between components.

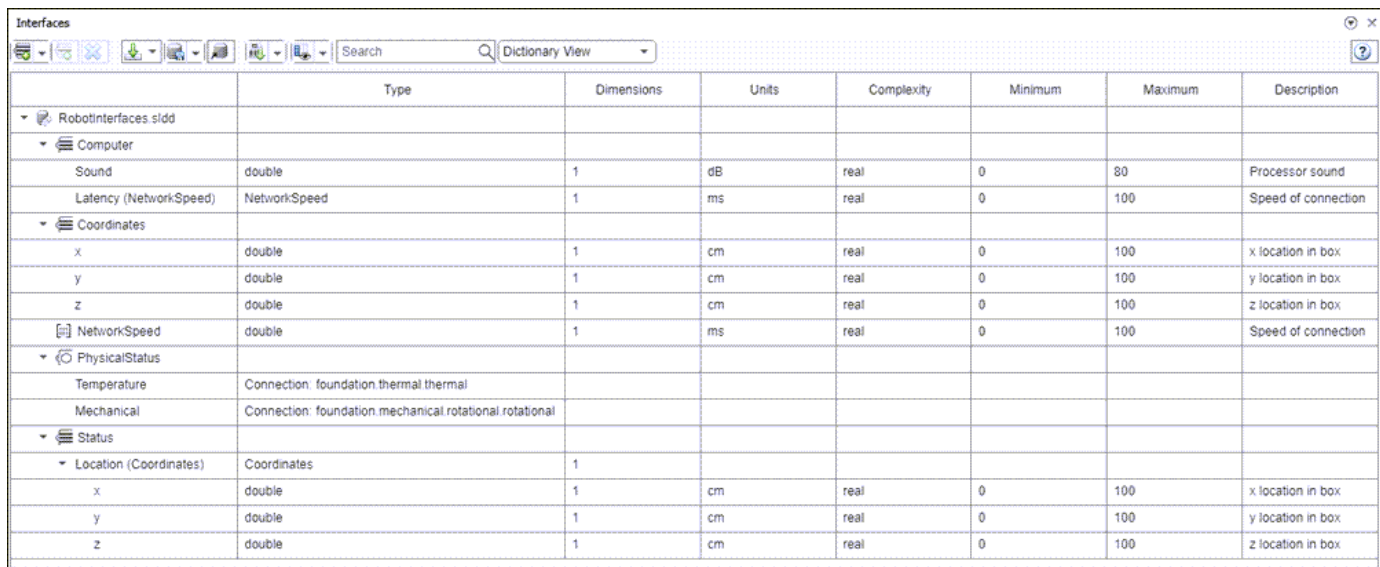
Types of interfaces include:

- **Composite Data Interface** — Represents the information that is shared through a connector and enters or exits a component through a port, A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.
- **Value Type** — Can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description. You can also assign the type of data elements in data interfaces to value types.
- **Physical Interface** — Defines the kind of information that flows through a physical port, The same interface can be assigned to multiple ports. A physical interface bundles physical elements to describe a physical model using at least one physical domain
- **Service Interface** — Defines service elements with function arguments for a client-server port. This interface is only available for software architectures.

You can save a locally defined model data dictionary as a shared data dictionary to reuse interface definitions across architecture models. Apply a profile to your interface dictionary to assign stereotypes to interfaces. These interfaces typed by a stereotype now contain metadata, and you can set the property values for each interface independently.

You can toggle the view for the **Interface Editor** depending on the locality of the interfaces:

- **Dictionary View** — Shows shared interfaces across the model that can be reused on multiple ports
- **Port Interface View** — Shows owned interfaces locally defined on a single port



	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
RobotInterfaces.sidd							
Computer							
Sound	double	1	dB	real	0	80	Processor sound
Latency (Network-Speed)	Network-Speed	1	ms	real	0	100	Speed of connection
Coordinates							
x	double	1	cm	real	0	100	x location in box
y	double	1	cm	real	0	100	y location in box
z	double	1	cm	real	0	100	z location in box
Network-Speed	double	1	ms	real	0	100	Speed of connection
PhysicalStatus							
Temperature	Connection: foundation thermal thermal						
Mechanical	Connection: foundation mechanical rotational rotational						
Status							
Location (Coordinates)	Coordinates	1					
x	double	1	cm	real	0	100	x location in box
y	double	1	cm	real	0	100	y location in box
z	double	1	cm	real	0	100	z location in box

## Open the Interface Editor

- System Composer toolstrip: Navigate to **Modeling > Interface Editor**.

## Examples

- “Modeling System Architecture of Small UAV”
- “Define Port Interfaces Between Components”
- “Specify Physical Interfaces on Ports”
- “Author Service Interfaces for Client-Server Communication”
- “Use Property Inspector in System Composer”

## Parameters


**Add data interface** — Add new data interface button

Add a new data interface by clicking  or select one of these options from the drop-down list:

- **Composite Data Interface** — Represents the information that is shared through a connector and enters or exits a component through a port. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.
- **Value Type** — Can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description. You can also assign the type of data elements in data interfaces to value types.
- **Physical Interface** — Defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface bundles physical elements to describe a physical model using at least one physical domain

- **Service Interface** — Defines service elements with function arguments for a client-server port. This interface is only available for software architectures.

**Add element to selected interface** — Add new element  
button

Add a new element by clicking . If the selected interface is one of these, the new element added is one of these types:

- **Composite Data Interface — Data Element**
- **Physical Interface — Physical Element**
- **Service Interface — Service Element — Function Arguments**, which are only available for software architectures

**Delete selected interface or element** — Delete interface or element  
button

Delete the selected interface or element in the **Interface Editor**.

**Import interfaces** — Import interface definitions  
button

Import interfaces from these locations:

- **Base Workspace**
- **MAT-file**

**Save interfaces and/or link dictionary** — Save interfaces or link dictionary  
button

Save interfaces on the current dictionary or link to an existing dictionary. Select a specific option from the drop-down list:

- **Save dictionary**
- **Save all dictionaries**
- **Save to new dictionary**
- **Link existing dictionary**

**Import profile** — Choose profile to import into data dictionary  
button

Choose a profile XML file to import into the currently selected data dictionary.

**Show Hide Columns** — Show and hide columns in editor  
button

Show and hide columns on the **Interface Editor** by checking the corresponding boxes:

- **Type**
- **Dimensions**
- **Units**

- **Complexity**
- **Minimum**
- **Maximum**
- **Description**
- **Asynchronous**, available only for software architectures

**View** — Choose editor view

**Dictionary View** (default) | **Port Interface View**

Choose a view for the **Interface Editor** to display interfaces:

- **Dictionary View** — Shows shared interfaces across the model that can be reused on multiple ports
- **Port Interface View** — Shows owned interfaces locally defined on a single port

## More About

### Interface Data Dictionary

An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.

Local interfaces on a System Composer model can be saved in an interface data dictionary using the **Interface Editor**. You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.

### Data Interface

A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.

Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the **Interface Editor** to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.

### Data Element

A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.

Data interfaces are decomposed into data elements:

- Pins or wires in a connector or harness.
- Messages transmitted across a bus.
- Data structures shared between components.

### Value Type

A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.

You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the **Interface Editor** so that you can reuse the value types as interfaces or data elements.

### **Owned Interface**

An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.

Create an owned interface to represent a value type or data interface that is local to a port.

### **Adapter**

An adapter helps connect two components with incompatible port interfaces by mapping between the two interfaces. An adapter can act as a unit delay or rate transition. You can also use an adapter for bus creation. Use the Adapter block to implement an adapter.

With an adapter, you can perform functions on the “Interface Adapter” dialog box:

- Create and edit mappings between input and output interfaces.
- Apply an interface conversion `UnitDelay` to break an algebraic loop.
- Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.
- Apply an interface conversion `Merge` to merges two or more message or signal lines.
- When output interfaces are undefined, you can use input interfaces in bus creation mode to author owned output interfaces.

### **Physical Interface**

A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a `Simulink.ConnectionBus` object that specifies any number of `Simulink.ConnectionElement` objects.

Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.

### **Physical Element**

A physical element describes the decomposition of a physical interface. A physical element is equivalent to a `Simulink.ConnectionElement` object.

Define the `Type` of a physical element as a physical domain to enable use of that domain in a physical model.

### **Function**

A function is an entry point that can be defined in a software component.

You can apply stereotypes to functions in software architectures, edit sample times, and specify the function period using the **Functions Editor**.



## Service Interface

A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.

Once you have defined a service interface in the **Interface Editor**, you can assign it to client and server ports using the **Property Inspector**. You can also use the **Property Inspector** to assign stereotypes to service interfaces.

## Function Element

A function element describes the attributes of a function in a client-server interface.

Edit the function prototype on a function element to change the number and names of inputs and outputs of the function. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects. You can specify function elements to support:

- Synchronous execution — When the client calls the server, the function runs immediately and returns the output arguments to the client.
- Asynchronous execution — When the client makes a request to call the server, the function is executed asynchronously based on the priority order defined in the **Functions Editor** and **Schedule Editor** and returns the output arguments to the client.

## Function Argument

A function argument describes the attributes of an input or output argument in a function element.

You can set the properties of a function argument in the **Interface Editor** just as you would any value type: Type, Dimensions, Units, Complexity, Minimum, Maximum, and Description.

# Version History

**Introduced in R2019a**

## See Also

`addInterface` | `removeInterface` | `addElement` | `removeElement` | `connect` | `setInterface` | `addValueType` | `connect` | `getDestinationElement` | `getSourceElement` | `createInterface` | `createOwnedType` | `Adapter` | `createDictionary` | `openDictionary` | `saveToDictionary` | `linkDictionary` | `unlinkDictionary` | `addReference` | `removeReference`

## Topics

“Modeling System Architecture of Small UAV”

“Define Port Interfaces Between Components”

“Specify Physical Interfaces on Ports”

“Author Service Interfaces for Client-Server Communication”

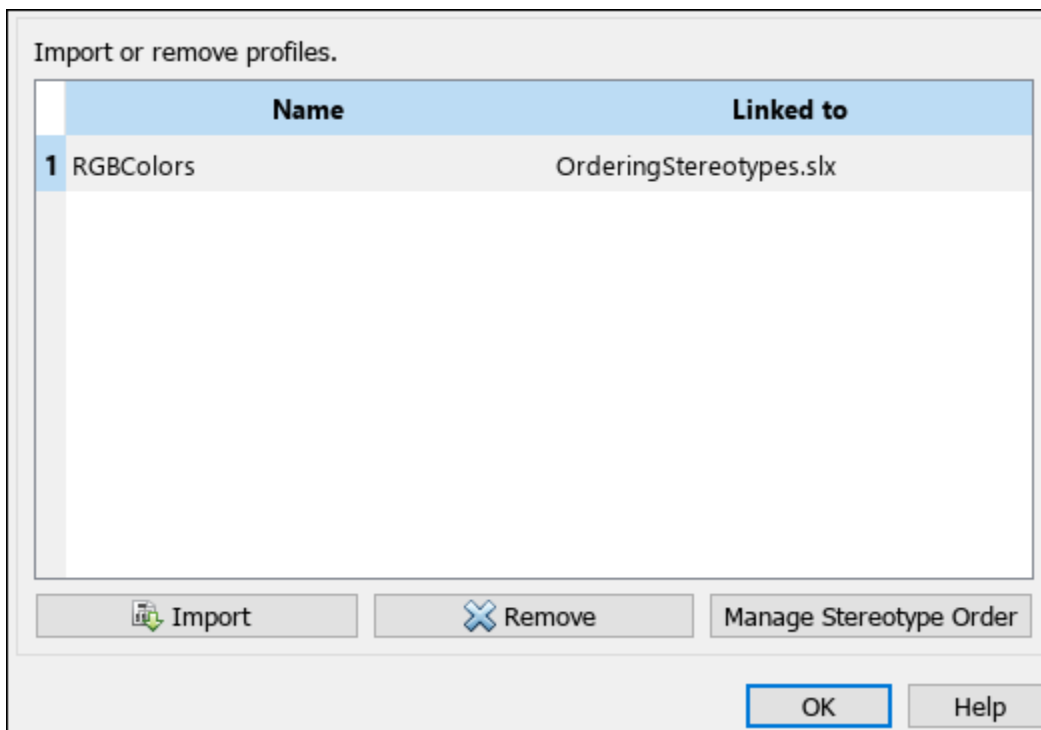
“Use Property Inspector in System Composer”

## Manage Profiles

Link and unlink profiles and order stereotypes

### Description

Use the **Manage Profiles** tool to import profiles into the current architecture model, or remove profiles from the model that have already been imported. The imported profiles appear in a list by name and the model or dictionary to which a profile is linked. To manage the priority order of the stereotypes from all imported profiles, click **Manage Stereotype Order**. To define and edit profiles, use the **Profile Editor** tool.



### Open the Manage Profiles

- System Composer toolstrip: Navigate to **Modeling > Profile Editor > Manage**.

### Examples

- “Change Stereotype Order Using Manage Profiles Tool”
- “Define Profiles and Stereotypes”
- “Use Stereotypes and Profiles”
- “Define Stereotypes and Perform Analysis”
- “Apply Stereotypes to Functions of Software Architectures”

## Parameters

**Import** — Import profile into model  
button

Import a profile into the current architecture model by navigating to the current directory and choosing a profile with an `.xml` extension.

**Remove** — Remove profile from model or dictionary  
button

Remove the selected profile on the list from the model or dictionary to which the profile is linked.

**Manage Stereotype Order** — Manage order of stereotypes for imported profiles  
button

Manage the priority order of stereotypes for imported profiles so that when multiple profiles are applied to a model element, the highest priority stereotype will display stereotype-based styling.

For more information, see “Change Stereotype Order Using Manage Profiles Tool”.

---

**Note** Connector styling is sourced from the highest-priority stereotype that defines style information. Connector stereotypes have the highest priority, followed by port stereotypes and then interface stereotypes. When two connectors with different styling merge, if the styling is incompatible, the resulting connector is displayed in black.

---

## Programmatic Use

`model.applyProfile(profile)` links a profile to the model.

`model.removeProfile(profile)` unlinks a profile from the model.

## More About

### Model

A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.

Perform operations on a model:

- Extract the root-level architecture contained in the model.
- Apply profiles.
- Link interface data dictionaries.
- Generate instances from model architecture.

A System Composer model is stored as an SLX file.

## Interface Data Dictionary

An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.

Local interfaces on a System Composer model can be saved in an interface data dictionary using the **Interface Editor**. You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.

## Profile

A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.

Author profiles and apply profiles to a model using the **Profile Editor**. You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.

## Stereotype

A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.

Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.

## Property

A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.

Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the **Property Inspector**.

## Component

A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.

Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:

- Port interfaces using the **Interface Editor**
- Parameters using the **Parameter Editor**

## Port

A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.

There are different types of ports:

- *Component ports* are interaction points on the component to other components.
- *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.

### Connector

Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.

A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.

## Version History

Introduced in R2019a

### See Also

#### Tools

Profile Editor

#### Objects

`systemcomposer.profile.Profile` | `systemcomposer.profile.Stereotype` | `systemcomposer.profile.Property`

#### Functions

`systemcomposer.profile.Profile.createProfile` | `addStereotype` | `addProperty` | `batchApplyStereotype` | `applyStereotype` | `applyProfile` | `removeProfile`

#### Topics

“Change Stereotype Order Using Manage Profiles Tool”

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

“Define Stereotypes and Perform Analysis”

“Apply Stereotypes to Functions of Software Architectures”

# Parameter Editor

Add, edit, and promote parameters for architectures and components

## Description

The **Parameter Editor** allows you to add intrinsic or operational parameters for architectural design.

Use the **Parameter Editor** to:

- Add and edit parameters for components in an architecture. Edit the default properties of the parameter: Name, Value, Unit, Type, Dimensions, Min, and Max
- Add and edit parameters to the root architecture of a model or to the architecture of a group of components.
- Promote parameters from components contained in the model to a top-level architecture.

Parameter Editor: Propeller

PARAMETERS EDIT

Highlight source Delete Cut Copy Paste ACTION

Controls Parameters & Dialog Property Editor

Search

PARAMETER

Add parameter Promote parameter...

Type	Prompt	Name
Parameters	Parameters	ParameterGroupVar
#1	advanceSpeed	advanceSpeed
#2	spinningRate	spinningRate
#3	bladePitch	bladePitch

Parameter Promotion: One-To-One

Search  Show Selected Promote

Name	Prompt
Propeller	
Hub	
bladePitch	bladePitch

PROPERTY EDITOR

PROPERTY	VALUE
Name	bladePitch
Prompt	bladePitch
Value	45
Unit	
Type	double
Dimens...	[1 1]
Min	
Max	
Type	edit
Source	Hub/bladePitch

## Open the Parameter Editor

- System Composer: From the **Property Inspector**, use the Parameters list to open the **Parameter Editor** using the Open Editor option.

## Examples

- “Author Parameters in System Composer Using Parameter Editor”
- “Use Parameters to Store Instance Values with Components”
- “Access Model Arguments as Parameters on Reference Components”
- “Use Property Inspector in System Composer”

## Parameters

**Add Parameter** — Add parameters to current architecture  
button

Add parameters to the current architecture. The architecture can be the root architecture of the model or the architecture of the currently selected component.

**Promote Parameters** — Open parameter promotion  
button

Open the parameter promotion user interface. If there are components with parameters in the currently selected architecture, you can promote these parameters by selecting each check box and clicking **Promote**.

**Highlight Source** — Highlight source of parameter  
button

Highlight the source of the parameter in the model canvas and bring it into the front view. To leave this spotlight view, click the close button at the top right of the model canvas.

## More About

### Parameter

A parameter is an instance-specific value of a value type.

Parameters are available for inlined architectures and components. Parameters are also available for components linked to model references or architecture references that specify model arguments. You can specify independent values for a parameter on each component.

### Component

A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.

Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:

- Port interfaces using the **Interface Editor**
- Parameters using the **Parameter Editor**

### **Architecture**

A System Composer architecture represents a system of components and how they interface with each other structurally and behaviorally.

Different types of architectures describe different aspects of systems. You can use views to visualize a subset of components in an architecture. You can define parameters on the architecture level using the **Parameter Editor**.

## **Version History**

**Introduced in R2022b**

### **See Also**

`systemcomposer.arch.Parameter` | `addParameter` | `getParameter` | `getParameterPromotedFrom` | `resetToDefault` | `getEvaluatedParameterValue` | `getParameterNames` | `setParameterValue` | `getParameterValue` | `setUnit` | `resetParameterToDefault`

### **Topics**

“Author Parameters in System Composer Using Parameter Editor”

“Use Parameters to Store Instance Values with Components”

“Access Model Arguments as Parameters on Reference Components”

“Use Property Inspector in System Composer”



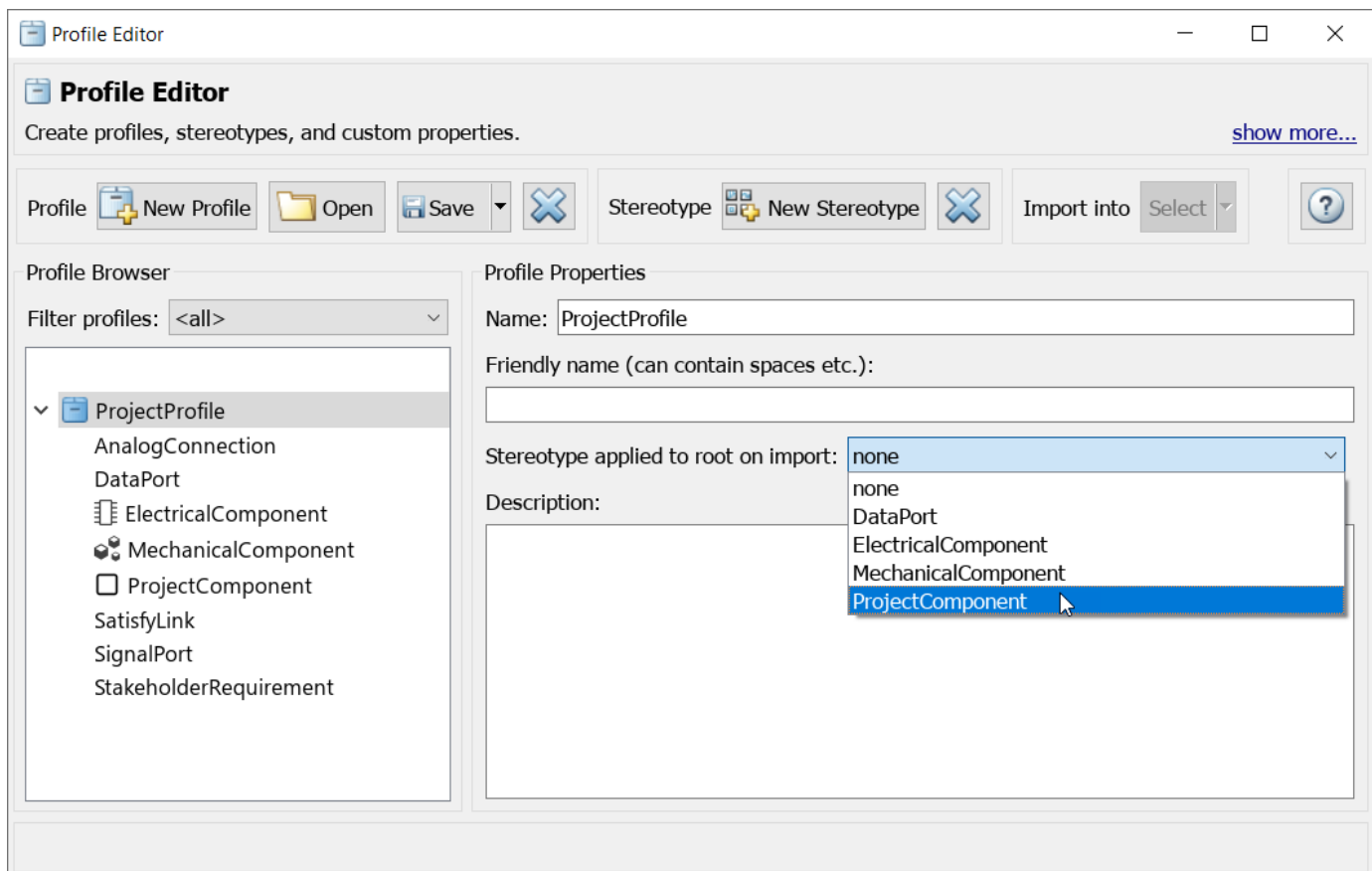
# Profile Editor

Create and manage profiles with stereotypes and properties

## Description

The **Profile Editor** allows you to define a profile that contains stereotypes with properties. In System Composer architecture models, stereotyping is necessary to define custom metadata on model elements typed by the stereotype. In Requirements Toolbox, you can use stereotypes to define custom requirement types and link types with custom properties.

- **System Composer:** Apply a profile to your model or interface data dictionary. Then, use stereotypes in the model to type model elements such as components, connectors, ports, interfaces, functions, requirement sets, and link sets. Functions only apply to software architectures. You can define custom property values on each element using the stereotyped template.
- **Requirements Toolbox:** Apply a profile to a requirement set or link set. Then use stereotypes by setting the requirement type or link type to the stereotype and setting the stereotype properties to your desired values.



## Open the Profile Editor

### System Composer

- System Composer toolstrip: In the **Modeling** tab, click **Profile Editor**.
- MATLAB Command Window: Enter `systemcomposer.profile.editor`.

### Requirements Toolbox

- **Requirements Editor** toolstrip: Click **Profile Editor** .

## Examples

- “Define Stereotypes and Perform Analysis”
- “Define Profiles and Stereotypes”
- “Use Stereotypes and Profiles”
- “Apply Stereotypes to Functions of Software Architectures”
- “Use Property Inspector in System Composer”
- “Customize Requirements and Links by Using Stereotypes” (Requirements Toolbox)

## Parameters

**Filter profiles** — Filter to show imported profiles

`<all>` (default) | model file name | dictionary file name | `<refresh>`

Filter imported profiles:

- `<all>` to show all imported profiles from all loaded models and dictionaries.
- A model name, such as `model.slx`, to show all imported profiles from specified architecture model.
- An interface data dictionary, such as `dictionary.sldd`, to show all imported profiles from specified interface data dictionary.
- `<refresh>` to refresh profiles from all loaded models and dictionaries.

**Import into** — Import selected profile

model file name | dictionary file name

Specify the name of a model or interface data dictionary to which to import the selected profile.

**Stereotype applied to root on import** — Root stereotype

`<none>` (default) | stereotype

Stereotype to apply to the root architecture after importing profile into a model. Choose from a list of available stereotypes. The root architecture is at the system boundary of the top-level model that separates the contents of the model from the environment.

**Applies to** — Element type to which stereotype can be applied

`<all>` (default) | Component | Port | Connector | Interface | Function | Requirement | Link

Element type to which the stereotype can be applied.

**Base stereotype** — Stereotype from which stereotype inherits properties  
 <none> (default) | stereotype

Stereotype from which the stereotype inherits properties. Choose from a list of available stereotypes.

**Abstract stereotype** — Whether stereotype is abstract  
 off (default) | on

Select this check box to indicate an abstract stereotype. An abstract stereotype is a stereotype that is not intended to be applied directly to a model element. You can use abstract stereotypes only as the base stereotype for other stereotypes.

**Show inherited properties** — Whether to show properties inherited from base stereotype  
 off (default) | on

Select this check box to indicate whether to display read-only properties inherited from a base stereotype.

## Programmatic Use

`systemcomposer.profile.editor` opens the **Profile Editor** from the MATLAB Command Window.

## More About

### Model

A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.

Perform operations on a model:

- Extract the root-level architecture contained in the model.
- Apply profiles.
- Link interface data dictionaries.
- Generate instances from model architecture.

A System Composer model is stored as an SLX file.

### Interface Data Dictionary

An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.

Local interfaces on a System Composer model can be saved in an interface data dictionary using the **Interface Editor**. You can reuse interface dictionaries between models that need to use a given set of interfaces, elements, and value types. Linked data dictionaries are stored in separate SLDD files.

## Profile

A profile is a package of stereotypes that you can use to create a self-consistent domain of element types.

Author profiles and apply profiles to a model using the **Profile Editor**. You can store stereotypes for a project in one or several profiles. When you save profiles, they are stored in XML files.

## Stereotype

A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.

Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, functions, requirements, and links. Functions only apply to software architectures. You must have a Requirements Toolbox license to apply stereotypes to requirements and links. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.

## Property

A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.

Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the **Property Inspector**.

## Component

A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. A component defines an architectural element, such as a function, a system, hardware, software, or other conceptual entity. A component can also be a subsystem or subfunction.

Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts. Transfer information between components with:

- Port interfaces using the **Interface Editor**
- Parameters using the **Parameter Editor**

## Port

A port is a node on a component or architecture that represents a point of interaction with its environment. A port permits the flow of information to and from other components or systems.

There are different types of ports:

- *Component ports* are interaction points on the component to other components.
- *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.

## Connector

Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.

A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.

## Data Interface

A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.

Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the **Interface Editor** to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.

## Physical Interface

A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to a `Simulink.ConnectionBus` object that specifies any number of `Simulink.ConnectionElement` objects.

Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.

## Service Interface

A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements.

Once you have defined a service interface in the **Interface Editor**, you can assign it to client and server ports using the **Property Inspector**. You can also use the **Property Inspector** to assign stereotypes to service interfaces.

## Requirements

Requirements are a collection of statements describing the desired behavior and characteristics of a system. Requirements ensure system design integrity and are achievable, verifiable, unambiguous, and consistent with each other. Each level of design should have appropriate requirements.

To enhance traceability of requirements, link system, functional, customer, performance, or design requirements to components and ports. Link requirements to each other to represent derived or allocated requirements. Manage requirements from the Requirements Perspective on an architecture model or through custom views. Assign test cases to requirements using the **Test Manager** for verification and validation.

## Requirement Link

A link is an object that relates two model-based design elements. A requirement link is a link where the destination is a requirement. You can link requirements to components or ports.

View links using the Requirements Perspective in System Composer. Select a requirement in the Requirements Browser to highlight the component or the port to which the requirement is assigned. Links are stored externally as SLMX files.

### **Requirement Set**

A requirement set is a collection of requirements. You can structure the requirements hierarchically and link them to components or ports.

Use the **Requirements Editor** to edit and refine requirements in a requirement set. Requirement sets are stored in SLREQX files. You can create a new requirement set and author requirements using Requirements Toolbox, or import requirements from supported third-party tools.

## **Version History**

**Introduced in R2019a**

### **See Also**

#### **Tools**

**Profile Editor**

#### **Objects**

`systemcomposer.profile.Profile` | `systemcomposer.profile.Stereotype` |  
`systemcomposer.profile.Property`

#### **Functions**

`systemcomposer.profile.editor` | `systemcomposer.profile.Profile.createProfile` |  
`addStereotype` | `addProperty`

#### **Topics**

“Define Stereotypes and Perform Analysis”

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

“Apply Stereotypes to Functions of Software Architectures”

“Use Property Inspector in System Composer”

“Customize Requirements and Links by Using Stereotypes” (Requirements Toolbox)

# Sequence Viewer

Visualize messages, events, states, transitions, and functions

## Description

The Sequence Viewer visualizes message flow, function calls, and state transitions.

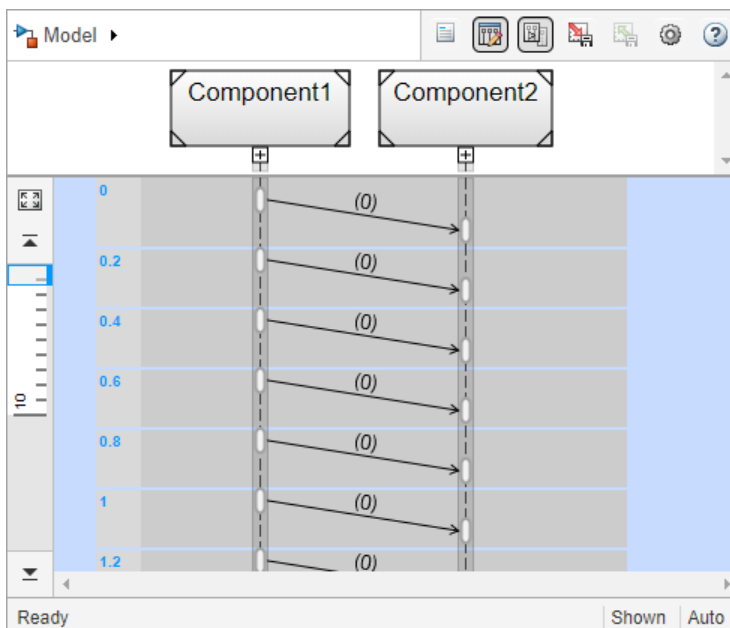
Use the Sequence Viewer to see the interchange of messages, events, function calls in Simulink models, Simulink behavior models in System Composer and between Stateflow charts in Simulink models.

In the Sequence Viewer window, you can view event data related to Stateflow chart execution and the exchange of messages between Stateflow charts. The Sequence Viewer window shows messages as they are created, sent, forwarded, received, and destroyed at different times during model execution. The Sequence Viewer window also displays state activity, transitions, and function calls to Stateflow graphical functions, Simulink functions, and MATLAB functions. For more information, see .

---

**Note** The Sequence Viewer does not display function calls generated by MATLAB Function blocks and S-functions.

---



## Open the Sequence Viewer

- Simulink Toolstrip: On the **Simulation** tab, in the **Review Results** section, click **Sequence Viewer**.

## Examples

### Using the Sequence Viewer Tool

- 1 To activate logging events, in the Simulink Toolstrip, under the **Simulation** tab, in the **Prepare** section, click **Log Events**.
  - 2 Simulate your model.
  - 3 To open the tool, in the Simulink Toolstrip, under the **Simulation** tab, in the **Review Results** section, click **Sequence Viewer**.
- “Simulink Messages Overview”

## Parameters

### Time Precision for Variable Step — Digits for time increment precision

3 (default) | scalar

Number of digits for time increment precision. When using a variable step solver, change this parameter to adjust the time precision for the sequence viewer. By default the block supports 3 digits of precision. Minimum and maximum precision are 1 and 16, respectively.

Suppose the block displays two events that occur at times 0.1215 and 0.1219. Displaying these two events precisely requires 4 digits of precision. If the precision is 3, then the block displays two events at time 0.121.

#### Programmatic Use

**Block Parameter:** SequenceViewerTimePrecision

**Type:** character vector

**Values:** '3' | scalar

**Default:** '3'

### History — Maximum number of previous events to display

1000 (default) | scalar

Total number of events before the last event to display. Minimum and maximum number of events are 0 and 25000, respectively.

For example, if **History** is 5 and there are 10 events in your simulation, then the block displays 6 events, including the last event and the five events prior the last event. Earlier events are not displayed. The time ruler is greyed to indicate the time between the beginning of the simulation and the time of the first displayed event.

Each send, receive, drop, or function call event is counted as one event, even if they occur at the same simulation time.

#### Programmatic Use

**Block Parameter:** SequenceViewerHistory

**Type:** character vector

**Values:** '1000' | scalar

**Default:** '1000'



## **Version History**

**Introduced in R2020b**

**Blocks**

**Topics**

“Simulink Messages Overview”

