

System Composer™

Reference



MATLAB® & SIMULINK®

R2021b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: 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 2019–2021 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 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 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)

1	<hr/>	Functions
2	<hr/>	Classes
3	<hr/>	Blocks

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 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 where variant choices are added, 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 Condition" on page 1-531

See Also

`getActiveChoice` | `getChoices` | `makeVariant` | `addVariantComponent` | Variant Component

Topics

"Create Variants"

Introduced in R2019a

addComponent

Package: systemcomposer.arch

Add components to architecture

Syntax

```
components = addComponent(architecture, compNames)
components = addComponent(architecture, compNames, stereotypes)
```

Description

`components = addComponent(architecture, compNames)` adds a set of components specified by the names `compNames`.

`components = addComponent(architecture, 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.

```
model = systemcomposer.createModel('archModel', true);
arch = get(model, 'Architecture');
names = {'Component1', 'Component2'};
comp = addComponent(arch, names);
```

Input Arguments

architecture — Parent architecture

architecture object

Parent architecture to add component to, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

addPort | connect | Component

Topics

“Components”

Introduced in R2019a

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.

Compatibility Considerations

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”

Introduced in R2019b

addElement

Package: systemcomposer.interface

Add data 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.

`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 named '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-2d2fcdbb5c25'
    ExternalUID: ''
```

Input Arguments

interface — Interface

data interface object | physical interface object

Interface to add the element to, specified as a `systemcomposer.interface.DataInterface` or `systemcomposer.interface.PhysicalInterface` object.

name — Name of new element

character vector | string

Name of new element, specified as a character vector or string with a valid MATLAB® variable name.

Data Types: char | string

Name-Value Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

```
addElement(interface, 'newElement', '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: `addElement(interface, 'newElement', '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: `addElement(interface, 'newElement', '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: `addElement(interface,'newElement','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: `addElement(interface,'newElement','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: `addElement(interface,'newElement','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: `addElement(interface,'newElement','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: `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 — New data element

data element object | physical element object

New data element, returned as a `systemcomposer.interface.DataElement` or `systemcomposer.interface.PhysicalElement` 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`removeElement` | `getElement` | `getInterfaceNames` | `getInterface` | `setType` | `addInterface` | `addValueType` | `addPhysicalInterface`

Topics

"Specify Physical Interfaces on the Ports"
 "Create Interfaces"
 "Manage Interfaces with Data Dictionaries"

Introduced in R2019a

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 addElement cannot be used when a query is defined on the view. To remove the query, run removeQuery.

Examples

Add Elements to View

Open up the key-less 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 named '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 a 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 add to 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.

The components do not need to be ordered.

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. <i>Functional views</i> focus on what the system must do to operate. <i>Physical views</i> show how the system is constructed and configured. <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"> "Create Architecture Views Interactively" "Create Architectural Views Programmatically"
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

openViews | createView | getView | deleteView | systemcomposer.view.ElementGroup | systemcomposer.view.View | removeElement | getSubGroup | deleteSubGroup | createSubGroup

Topics

“Create Architecture Views Interactively”
 “Create Architectural Views Programmatically”

Introduced in R2021a

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`.

`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 named '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');

% Create the Simulink bus object and populate it with elements
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',1);  
linkDictionary(arch,'new_dictionary.sldd');
```

Input Arguments

dictionary — Data dictionary attached to architecture model

dictionary object

Data dictionary attached to architecture model, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`addElement` | `createDictionary` | `getInterface` | `getInterfaceNames` | `removeInterface` | `linkDictionary` | `Adapter` | `addPhysicalInterface` | `addValueType`

Topics

“Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2019a

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`.

Examples

Add Physical Interface

Create a data dictionary, then add a physical interface named '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 attached to architecture model

dictionary object

Data dictionary attached to architecture model, 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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`addElement` | `createDictionary` | `addInterface` | `getInterface` | `getInterfaceNames` | `removeInterface` | `linkDictionary` | `Adapter` | `addValueType`

Topics

“Specify Physical Interfaces on the Ports”
 “Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2021b

addPort

Package: systemcomposer.arch

Add ports to architecture

Syntax

```
ports = addPort(architecture, portNames, portTypes)
ports = addPort(architecture, portNames, portTypes, stereotypes)
```

Description

`ports = addPort(architecture, portNames, portTypes)` adds a set of ports with specified names `portNames` and types `portTypes`.

`ports = addPort(architecture, 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.SignalInterface]
    Connectors: [0x0 systemcomposer.arch.Connector]
    Connected: 0
    Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 52.0001
    SimulinkModelHandle: 49.0001
    UUID: '98070dc5-1738-4dbf-b9b2-4fc781e7992c'
    ExternalUID: ''
```

Input Arguments

architecture — Architecture

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object. `addPort` 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.

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"

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none">• “Define Profiles and Stereotypes”• “Use Stereotypes and Profiles”

See Also

destroy | systemcomposer.arch.BasePort | addComponent | connect | Component

Topics

“Ports”

Introduced in R2019a

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.

`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 to which property is added

stereotype object

Stereotype to which property is added, 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: '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

Dimensions — Dimensions of property

positive integer array

Dimensions of property, specified as a positive integer array. Empty implies no restriction.

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)`

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)`

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”.

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

[getProperty](#) | [setProperty](#) | [removeProperty](#)

Topics

“Define Profiles and Stereotypes”

“Set Properties for Analysis”

Introduced in R2019a

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 an interface named 'newInterface' to the local interface dictionary of the model. Save the local interface dictionary to a shared dictionary as an .sldd file.

```
% Create a new model and add an interface to its local dictionary  
arch = systemcomposer.createModel('newModel',true);  
addInterface(arch.InterfaceDictionary,'newInterface');
```

```
% Save interfaces from a local dictionary to a shared dictionary  
saveToDictionary(arch,'TopDictionary')
```

```
% Open the shared dictionary  
topDictionary = systemcomposer.openDictionary('TopDictionary.sldd');
```

Create a new dictionary and add it as a reference to the existing dictionary.

```
% Create a new dictionary  
refDictionary = systemcomposer.createDictionary('ReferenceDictionary.sldd');
```

```
% Add the new dictionary as a reference  
addReference(topDictionary,'ReferenceDictionary.sldd')
```

Input Arguments

dictionary – Dictionary

dictionary object

Dictionary, specified as a `systemcomposer.interface.Dictionary` object.

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

saveToDictionary | createDictionary | openDictionary | linkDictionary | unlinkDictionary | removeReference

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021a

addStereotype

Package: systemcomposer.profile

Add stereotype to profile

Syntax

```
stereotype = addStereotype(profile, stereotypeName)
stereotype = addStereotype( ____, Name, Value)
```

Description

`stereotype = addStereotype(profile, stereotypeName)` adds a new stereotype with a specified `stereotypeName` to a 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: 'default'
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]
```

Input Arguments

profile – Profile object

profile

Profile object, specified as a `systemcomposer.profile.Profile` object.

stereotypeName — Name of new stereotype

character vector | string

Name of new stereotype, specified as a character vector or string. The name of the stereotype must be unique within the profile.

Data Types: char | string

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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.

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'default')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'application')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'channel')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'controller')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'database')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'devicedriver')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'memory')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'network')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'plant')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'sensor')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'subsystem')`

Example: `addStereotype(profile, 'electricalComponent', 'Icon', 'transmitter')`

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

'Component' | 'Port' | 'Connector' | 'Interface'

Element type to which stereotype can be applied, specified as a character vector or string of the following options:

- 'Component'
- 'Port'
- 'Connector'
- 'Interface'

Example: `addStereotype(profile, 'electricalComponent', 'AppliesTo', 'Port')`

Data Types: `char` | `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.

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].

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].

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.

Example:

`addStereotype(profile, 'electricalComponent', 'ConnectorLineStyle', 'Default')`

Example:

```
addStereotype(profile, 'electricalComponent', 'ConnectorLineStyle', 'Dot')
```

Example:

```
addStereotype(profile, 'electricalComponent', 'ConnectorLineStyle', 'Dash')
```

Example: `addStereotype(profile, 'electricalComponent', 'ConnectorLineStyle', 'Dash Dot')`

Example: `addStereotype(profile, 'electricalComponent', 'ConnectorLineStyle', 'Dash Dot Dot')`

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none">• “Define Profiles and Stereotypes”• “Use Stereotypes and Profiles”

See Also

`getStereotype` | `getDefaultStereotype` | `setDefaultStereotype` | `removeStereotype`

Topics

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

Introduced in R2019a

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.

`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 named 'airSpeed'.

```
dictionary = systemcomposer.createDictionary('new_dictionary.sldd');
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.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 one 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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 — New value type

value type object

New 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> “Manage Interfaces with Data Dictionaries” “Reference Data Dictionaries”

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 Interface Editor to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

addElement | createDictionary | getInterface | getInterfaceNames | removeInterface | linkDictionary | Adapter | addPhysicalInterface | addInterface

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021b

addVariantComponent

Package: systemcomposer.arch

Add variant components to architecture

Syntax

```
variantList = addVariantComponent(architecture,variantComponents)
variantList = addVariantComponent(architecture,variantComponents,'Position',
position)
```

Description

`variantList = addVariantComponent(architecture,variantComponents)` adds a set of components specified by the cell array of names.

`variantList = addVariantComponent(architecture,variantComponents,'Position',position)` creates a variant component the architecture at a given position.

Examples

Create Variant with Two Components

Create model, get root architecture, and create a component with two variants.

```
model = systemcomposer.createModel('archModel',true);
arch = get(model,'Architecture');
names = {'Component1','Component2'}
variants = addVariantComponent(arch,names);
```

Input Arguments

architecture — Parent architecture

architecture object

Parent architecture to which component is added, 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

variantList – 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Condition" on page 1-531

See Also

addPort | connect | addChoice | getActiveChoice | setActiveChoice | Variant Component

Topics

"Create Variants"

Introduced in R2019a

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`.

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% Save the allocation set
allocSet.save;

% Open the allocation editor
systemcomposer.allocation.editor()
```

Input Arguments

allocScenario — Allocation scenario

allocation scenario object

Allocation scenario in which to create allocations, specified as a `systemcomposer.allocation.AllocationScenario` object.

sourceElement — Source element for allocation

element object

Source element for allocation, 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 for allocation

element object

Target element for allocation, 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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

getAllocation | getAllocatedFrom | getAllocatedTo | deallocate | destroy | getScenario | createAllocationSet

Topics

“Create and Manage Allocations”

Introduced in R2020b

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` function and the `createView` function 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.

```
scKeylessEntrySystem
```

Open the model.

```
m = systemcomposer.openModel('KeylessEntryArchitecture');
```

Create a query to find all components and list the second.

```
constraint = AnyComponent();
components = find(m,constraint,'Recurse',true,'IncludeReferenceModels',true);
components(2)
```

```
ans =
```

```
1x1 cell array
```

```
{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller'}
```

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	"Display Component Hierarchy and Architecture Hierarchy Using Views"

See Also

`createView` | `find` | `systemcomposer.query.Constraint`

Topics

"Create Architectural Views Programmatically"

Introduced in R2019b

applyProfile

Package: systemcomposer.arch

Apply profile to model

Syntax

```
applyProfile(modelObject,profileFile)
```

Description

`applyProfile(modelObject,profileFile)` applies a profile to an architecture model and makes all the constituent stereotypes available.

Examples

Apply Profile

Create a model.

```
model = systemcomposer.createModel('archModel',true);
```

Create a profile with a stereotype, 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

modelObject — 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

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • "Set Properties" • "Add Properties with Stereotypes" • "Set Properties for Analysis"

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none">• “Define Profiles and Stereotypes”• “Use Stereotypes and Profiles”

See Also

`removeProfile` | `createProfile`

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

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, and signal interface model elements.

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, 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');  
  
model.applyProfile('LatencyProfile');
```

Apply the stereotype to the component, open the profile editor, and get the stereotypes on the component.

```
comp.applyStereotype('LatencyProfile.LatencyBase');
```

```
systemcomposer.profile.editor()
```

```
profile.save
```

```
stereotypes = getStereotypes(comp)
```

```
stereotypes =
```

```
    1x1 cell array
```

```
{'LatencyProfile.LatencyBase'}
```

Input 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, 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.interface.DataInterface`, `systemcomposer.ValueType`, or `systemcomposer.interface.PhysicalInterface` 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

[batchApplyStereotype](#) | [removeStereotype](#) | [getStereotypes](#) | [getStereotypeProperties](#)

Topics

"Use Stereotypes and Profiles"

Introduced in R2019a

batchApplyStereotype

Package: `systemcomposer.arch`

Apply stereotype to all elements in architecture

Syntax

```
batchApplyStereotype(architecture,elementType,stereotype)
batchApplyStereotype(architecture,elementType,stereotype,'Recurse',flag)
```

Description

`batchApplyStereotype(architecture,elementType,stereotype)` applies the stereotype to all elements that match the `elementType` within the architecture.

`batchApplyStereotype(architecture,elementType,stereotype,'Recurse',flag)` applies the stereotype to all elements that match the `elementType` within the architecture and its sub-architectures.

Examples

Apply a Stereotype to All Connectors

Apply the `standardConn` stereotype in the `GeneralProfile` profile to all connectors within the architecture `arch`.

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

Input Arguments

architecture — Architecture model element

architecture object

Architecture model element, specified as a `systemcomposer.arch.Architecture` object. Parent architecture layer for all components to attach the stereotype.

elementType — Type of architecture element

'Component' | 'Port' | 'Connector' | 'Instance'

Type of architecture element to apply the stereotype, specified as a character vector or string of 'Component', 'Port', 'Connector', or 'Instance'. The stereotype must be applicable for this element type.

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>`'. The stereotype must be applicable to components.

Data Types: char | string

flag – Apply stereotype recursively

false or 0 (default) | true or 1

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none">• “Set Properties”• “Add Properties with Stereotypes”• “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none">• “Define Profiles and Stereotypes”• “Use Stereotypes and Profiles”

See Also

`removeStereotype` | `applyStereotype` | `getStereotypes`

Topics

“Use Stereotypes and Profiles”

Introduced in R2019a

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.

```
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');
```

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

`systemcomposer.profile.Profile` | `open` | `editor` | `load` | `find` | `closeAll` | `save`

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

close

Package: systemcomposer.arch

Close model

Syntax

```
close(objModel)
```

Description

`close(objModel)` closes the specified model in System Composer.

Examples

Create, Open, and Close Model

```
model = systemcomposer.createModel('modelName');
open(model)
close(model)
```

Input Arguments

objModel — Model to close in editor

model object

Model to close in editor, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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”

See Also

createModel | save | loadModel

Topics

“Create an Architecture Model”

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% 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 the close

false or 0 (default) | true or 1

Force close the allocation set, specified as a logical or numeric value 1 (true) or 0 (false).

Data Types: `logical`

More About

Definitions

Term	Definition	Application	More Information
allocation	An allocation is a directed relationship from an element in one model to an element 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.	"Allocate Architectures in Tire Pressure Monitoring System"
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1 .	"Create and Manage Allocations"
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

`createScenario` | `deleteScenario` | `getScenario` | `load` | `closeAll` | `synchronizeChanges`

Topics

"Create and Manage Allocations"

Introduced in R2020b

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% 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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`createScenario` | `deleteScenario` | `getScenario` | `load` | `close` | `synchronizeChanges` | `find`

Topics

“Create and Manage Allocations”

Introduced in R2020b

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.

```
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');
```

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

systemcomposer.profile.Profile | load | editor | open | find | close | save

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

connect

Package: systemcomposer.arch

Create architecture model connections

Syntax

```
connectors = connect(srcComponent, destComponent)
connectors = connect(architecture, [srcComponent, srcComponent, ...], [
destComponent, destComponent, ...])
connectors = connect(architecture, [], destComponent)
connectors = connect(architecture, 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.

`connectors = connect(architecture, [srcComponent, srcComponent, ...], [destComponent, destComponent, ...])` connects arrays of components in the architecture.

`connectors = connect(architecture, [], destComponent)` connects a parent architecture input port to a destination child component.

`connectors = connect(architecture, 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

architecture — Architecture

architecture object

Architecture, specified as a `systemcomposer.arch.Architecture` object. The architecture is the interface and underlying structural definition of the model or component.

srcComponent — Source component

component object

Source component, specified as a `systemcomposer.arch.Component` object.

destComponent — Destination component

component object

Destination component, specified as a `systemcomposer.arch.Component` 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 as a character vector or string in the form '`<profile>.<stereotype>`'.

Data Types: `char` | `string`

Name-Value Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `connect(archPort, compPort, 'SourceElement', 'a')`

Stereotype — Option to apply stereotype to connector

character vector | string

Option to apply stereotype to connector, specified as a character vector or string 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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”

See Also

`openModel` | `createModel` | `addPort` | `getPort` | `addComponent` | `addElement` | `addInterface` | `setInterface` | `getSourceElement` | `getDestinationElement` | `Component`

Topics

“Connections”

“Build Architecture Models Programmatically”

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    'Source_Model_Allocation','Target_Model_Allocation');

% Save the allocation set
allocSet.save;

% 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

Term	Definition	Application	More Information
allocation	An allocation is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`load` | `open` | `closeAll`

Topics

“Create and Manage Allocations”

Introduced in R2020b

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.

Compatibility Considerations

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”

Introduced in R2019a

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`.

`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 a component named 'robotComp' in `Robot.slx` and reference the model.

Create a model '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 a component named 'robotComp' in `Robot.slx` and reference the model.

Create a model '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 a software architecture model so the component references the model Robot.slx.

```
createArchitectureModel(comp(2), 'Robot', 'SoftwareArchitecture');
```

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 — Model name

character vector | string

Model name, specified as a character vector or string.

Example: 'Robot'

Data Types: char | string

modelType — Type of model

'Architecture' (default) | 'SoftwareArchitecture'

Type of model to save, specified as 'Architecture' for an architecture model or 'SoftwareArchitecture' for a software 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> <i>Functional architecture</i> describes the flow of data in a system. <i>Logical architecture</i> describes the intended operation of a system. <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • “Describe Component Behavior Using Simulink” • “Create Reference Architecture”
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"> • “Create Simulink Behavior Using Simulink Subsystem” • “Create a Simulink Subsystem Component”
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"> • “Implement Behaviors for Architecture Model Simulation” • “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> • “Describe System Behavior Using Sequence Diagrams” • “Use Sequence Diagrams with Architecture Models”

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"> “Author Software Architectures” “Simulate and Deploy Software Architectures”
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"> “Implement Behaviors for Architecture Model Simulation” “Create Software Architecture from Component”
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 the Software Architecture of a Throttle Position Control System”
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”

See Also

[inlineComponent](#) | [createSubsystemBehavior](#) | [createSimulinkBehavior](#) | [createStateflowChartBehavior](#) | [extractArchitectureFromSimulink](#) | [linkToModel](#) | [isReference](#) | [Reference Component](#)

Topics

[“Describe Component Behavior Using Simulink”](#)
[“Decompose and Reuse Components”](#)
[“Describe Component Behavior Using Stateflow Charts”](#)
[“Create Simulink Behavior Using Simulink Subsystem”](#)
[“Simulate and Deploy Software Architectures”](#)

Introduced in R2021b

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.slidd')
```

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 `.slidd` extension and must be a valid MATLAB identifier.

Example: 'new_dictionary.slidd'

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"
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.	"Create an Architecture Model with Interfaces and Requirement Links"

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

[addValueType](#) | [addInterface](#) | [linkDictionary](#) | [saveToDictionary](#) | [unlinkDictionary](#) | [openDictionary](#) | [addReference](#) | [removeReference](#)

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2019a

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 named 'archModel'. Get the root architecture, then add a new component named 'newComponent' and a new port named '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'
    ExternalUID: ''
```

Create Owned Interface as Data Interface and Remove Owned Interface

Create an architecture model named 'archModel'. Get the root architecture, then add a new component named 'newComponent' and a new port named 'newCompPort'. Create an owned interface for the port as a 'DataInterface'. Remove the owned interface from the port.


```

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 named 'archModel'. Get the root architecture, then add a new component named 'newComponent' and a new physical port named '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

character vector | string

Kind of interface, specified as a character vector or string with the options 'DataInterface', 'ValueType', or 'PhysicalDomain'.

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"
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.	"Create an Architecture Model with Interfaces and Requirement Links"

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`addValueType` | `createModel` | `addInterface` | `setType` | `createOwnedType` | `addPhysicalInterface` | `removeInterface`

Topics

"Specify Physical Interfaces on the Ports"
 "Create Interfaces"
 "Manage Interfaces with Data Dictionaries"

Introduced in R2021b

createOwnedType

Package: systemcomposer.interface

Create owned type on data element

Syntax

```
ownedType = createOwnedType(dataElement)
ownedType = createOwnedType(dataElement, Name, Value)
```

Description

`ownedType = createOwnedType(dataElement)` creates an owned type on a data element defined as a value type.

`ownedType = createOwnedType(dataElement, Name, Value)` creates an owned type on a data element with additional options available for value types.

Examples

Create Owned Value Type 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'
ExternalUUID: ''
```

Select the architecture port 'inPort' on the architecture model and open the Property Inspector from the **Modeling > Design** menu. Select the edit link under **Open in Interface Editor**. Enter

the Port Interface View in the Interface Editor. Observe the new data element named 'newElement' under the port 'inPort'.

Input Arguments

dataElement — Data element

data element object

Data element, specified as a `systemcomposer.interface.DataElement` object.

Name-Value Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

```
createOwnedType(dataElement, '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: `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 type**

data interface object | value type object

Owned type, returned as a `systemcomposer.interface.DataInterface` or `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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. <p>System Composer models are stored as SLX files.</p>	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`addValueType` | `createModel` | `addInterface` | `setType` | `createInterface` | `removeInterface`

Topics

“Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2021b

systemcomposer.createModel

Create System Composer model

Syntax

```
objModel = systemcomposer.createModel(modelName)
objModel = systemcomposer.createModel(modelName,openFlag)
objModel = systemcomposer.createModel(modelName,modelType,openFlag)
```

Description

`objModel = systemcomposer.createModel(modelName)` creates a System Composer model with name `modelName` and returns the `systemcomposer.arch.Model` object.

`createModel` is the constructor method for the class `systemcomposer.arch.Model`.

`objModel = 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`.

`objModel = 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

objModel – Model

model object

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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"> • “Author Software Architectures” • “Simulate and Deploy Software Architectures”
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"> • “Implement Behaviors for Architecture Model Simulation” • “Create Software Architecture from Component”
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 the Software Architecture of a Throttle Position Control System”
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”

See Also

open | loadModel | save

Topics

“Compose Architecture Visually”

Introduced in R2019a

systemcomposer.profile.Profile.createProfile

Create profile

Syntax

```
profile = systemcomposer.profile.Profile.createProfile(profileName,dirPath)
profile = systemcomposer.profile.Profile.createProfile(profileName)
```

Description

`profile = systemcomposer.profile.Profile.createProfile(profileName,dirPath)` creates a new profile object `systemcomposer.profile.Profile` to add a set of stereotypes. The `dirPath` argument specifies the directory in which the profile is to be created.

`profile = systemcomposer.profile.Profile.createProfile(profileName)` creates a new profile with name `profileName`.

Examples

Create Profile

Create a model.

```
model = systemcomposer.createModel('archModel');
```

Create a profile with a stereotype, 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 new profile, specified as a character vector or string. This name must be a valid MATLAB identifier.

Example: 'LatencyProfile'

Data Types: char | string

dirPath – Directory path

character vector | string

Directory path where the profile will be saved, specified as a character vector.

Example: 'C:\Temp\MATLAB'

Data Types: char | string

Output Arguments

profile – Profile

profile object

Profile created, 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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

`applyProfile` | `loadProfile` | `editor` | `removeProfile` | `save` | `load` | `open` | `find`

Topics

“Create a Profile and Add Stereotypes”

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    'Source_Model_Allocation','Target_Model_Allocation');

% Get the default allocation scenario
defaultScenario = allocSet.getScenario('Scenario 1');

% Create a new allocation scenario
newScenario = allocSet.createScenario('Scenario 2');

% Save the allocation set
allocSet.save;

% 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 new allocation scenario

character vector | string

Name of new allocation scenario, specified as a character vector or string.

Example: 'Scenario 2'

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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`deleteScenario` | `getScenario` | `synchronizeChanges` | `load` | `closeAll` | `find` | `close`

Topics

“Create and Manage Allocations”

Introduced in R2020b

createSimulinkBehavior

Package: systemcomposer.arch

Create Simulink behavior and link to component

Syntax

```
createSimulinkBehavior(component,modelName)
```

Description

`createSimulinkBehavior(component,modelName)` creates a new Simulink model, `modelName`, with the same interface as the component `component` and links the component to the new model. The component must have no children.

Note To add Simulink subsystem behavior to a component, use the `createSubsystemBehavior` function. This behavior is part of the parent System Composer architecture model.

Examples

Create Simulink Model and Link to Component

Create a Simulink behavior model for the component named 'robotComp' in `Robot.slx` and link the model to the component.

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);
```

Create a Simulink behavior model for the 'robotComp' component so the component references the Simulink model `Robot.slx`.

```
createSimulinkBehavior(comp(2),'Robot');
```

Input Arguments

component — Architecture component

component object

Architecture component with no children, specified as a `systemcomposer.arch.Component` object.

modelName — Model name

character vector | string

Model name of the Simulink model to be created, specified as a character vector or string.

Example: 'Robot'

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • "Describe Component Behavior Using Simulink" • "Create Reference Architecture"

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"> • “Create Simulink Behavior Using Simulink Subsystem” • “Create a Simulink Subsystem Component”
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"> • “Implement Behaviors for Architecture Model Simulation” • “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> • “Describe System Behavior Using Sequence Diagrams” • “Use Sequence Diagrams with Architecture Models”

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"> • “Author Software Architectures” • “Simulate and Deploy Software Architectures”

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"> • “Implement Behaviors for Architecture Model Simulation” • “Create Software Architecture from Component”
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 the Software Architecture of a Throttle Position Control System”
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”

See Also

`inlineComponent` | `createArchitectureModel` | `createStateflowChartBehavior` | `createSubsystemBehavior` | `extractArchitectureFromSimulink` | `linkToModel` | `isReference` | Reference Component

Topics

- “Describe Component Behavior Using Simulink”
- “Decompose and Reuse Components”
- “Describe Component Behavior Using Stateflow Charts”
- “Create Simulink Behavior Using Simulink Subsystem”
- “Simulate and Deploy Software Architectures”

Introduced in R2019a

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.

Examples

Add Stateflow Chart Behavior to Component

Add Stateflow chart behavior to the component named 'robotComp' within the current model.

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));
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • “Describe Component Behavior Using Simulink” • “Create Reference Architecture”
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"> • “Create Simulink Behavior Using Simulink Subsystem” • “Create a Simulink Subsystem Component”

Term	Definition	Application	More Information
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"> • “Implement Behaviors for Architecture Model Simulation” • “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> • “Describe System Behavior Using Sequence Diagrams” • “Use Sequence Diagrams with Architecture Models”

See Also

`inlineComponent` | `createSimulinkBehavior` | `createArchitectureModel` | `createSubsystemBehavior` | `extractArchitectureFromSimulink` | `linkToModel` | `isReference` | Reference Component

Topics

- “Describe Component Behavior Using Simulink”
- “Decompose and Reuse Components”
- “Describe Component Behavior Using Stateflow Charts”
- “Create Simulink Behavior Using Simulink Subsystem”
- “Simulate and Deploy Software Architectures”

Introduced in R2021a

createSubsystemBehavior

Package: systemcomposer.arch

Add subsystem behavior to component

Syntax

```
createSubsystemBehavior(component)
```

Description

`createSubsystemBehavior(component)` adds subsystem 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 To create a new Simulink model with the same interface as the component and link the component to the new model, use the `createSimulinkBehavior` function.

Examples

Add Subsystem Behavior to Component

Add subsystem behavior to the component named 'robotComp' within the current model.

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 subsystem behavior model to the 'robotComp' component.

```
createSubsystemBehavior(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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • “Describe Component Behavior Using Simulink” • “Create Reference Architecture”
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"> • “Create Simulink Behavior Using Simulink Subsystem” • “Create a Simulink Subsystem Component”

Term	Definition	Application	More Information
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"> • “Implement Behaviors for Architecture Model Simulation” • “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> • “Describe System Behavior Using Sequence Diagrams” • “Use Sequence Diagrams with Architecture Models”

See Also

inlineComponent | createSimulinkBehavior | createArchitectureModel | createStateflowChartBehavior | extractArchitectureFromSimulink | linkToModel | isReference | Reference Component

Topics

“Describe Component Behavior Using Simulink”
 “Decompose and Reuse Components”
 “Describe Component Behavior Using Stateflow Charts”
 “Create Simulink Behavior Using Simulink Subsystem”
 “Simulate and Deploy Software Architectures”

Introduced in R2021b

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 `createSubGroup` 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 named 'NewView'.

```
model.openViews
```

Create a subgroup.

```
group = view.Root.createSubGroup('MyGroup')
```

```
group =
```

```
    ElementGroup with properties:
```

```
        Name: 'MyGroup'
        UUID: '46eaaed7-3ba0-418e-bc65-1ef8bce3087b'
        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.

Data Types: char | string

Output Arguments**subGroup — Subgroup**

element group object

Subgroup, returned as a `systemcomposer.view.ElementGroup` 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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. <i>Functional views</i> focus on what the system must do to operate. <i>Physical views</i> show how the system is constructed and configured. <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"> “Create Architecture Views Interactively” “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

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”

Introduced in R2021a

createView

Package: systemcomposer.arch

Create architecture view

Syntax

```
view = createView(model,viewName)
view = createView( ____,Name,Value)
```

Description

`view = createView(model,viewName)` creates a new architecture view `view` for the System Composer model `model` with the specified name `viewName`.

`view = createView(____,Name,Value)` creates a new view with additional options.

Examples

Create New 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.

```
scKeylessEntrySystem
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 — Model

model object

Model, specified as a `systemcomposer.arch.Model` object.

viewName — Name of new view

character vector | string

Name of new 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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

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 comma-separating pair consisting of 'Select' and a `systemcomposer.query.Constraint` object. A constraint can contain a sub-constraint 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.
HasPort	A component has a port that satisfies the given sub-constraint.
HasInterface	A port has an interface that satisfies the given sub-constraint.
HasInterfaceElement	An interface has an interface element that satisfies the given sub-constraint.
HasStereotype	An architecture element has a stereotype that satisfies the given sub-constraint.
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 comma-separating pair consisting of 'GroupBy' and 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:

`{'AutoProfile.MechanicalComponent.mass', 'AutoProfile.MechanicalComponent.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 the comma-separated pair consisting of 'IncludeReferenceModels' and a logical `1` (`true`) to search for referenced architectures or `0` (`false`) to not include referenced architectures.

Example: 'IncludeReferenceModels',`false`

Data Types: `logical`

Color — Color of view

character array | `string`

Color of view, specified as the comma-separated pair consisting of 'Color' and a character array or `string` that contains the name of the color or an RGB hexadecimal value.

Example: 'Color', 'blue'

Example: '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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

systemcomposer.view.View | getView | deleteView | openViews | systemcomposer.view.ElementGroup

Topics

“Create Architecture Views Interactively”
 “Create Architectural Views Programmatically”

Introduced in R2021a

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(object, name)
view = createViewArchitecture(object, name, constraint)
view = createViewArchitecture(object, name, constraint, groupBy)
view = createViewArchitecture( ____, Name, Value)
```

Description

`view = createViewArchitecture(object, name)` creates an empty view with the given name and default color 'blue'.

`view = createViewArchitecture(object, 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(object, 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

object — Model

architecture model object

Model to use to create a view, 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 sub-constraint that can be joined together with another constraint using AND or OR. A constraint can also 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.
HasPort	A component has a port that satisfies the given sub-constraint.
HasInterface	A port has an interface that satisfies the given sub-constraint.
HasInterfaceElement	An interface has an interface element that satisfies the given sub-constraint.
HasStereotype	An architecture element has a stereotype that satisfies the given sub-constraint.
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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

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.

Compatibility Considerations**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”

Introduced in R2019b

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.

Compatibility Considerations

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”

Introduced in R2019b

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% Deallocate components between models
defaultScenario.deallocate(sourceComp,targetComp);

% Save the allocation set
allocSet.save;

% Open the allocation editor
systemcomposer.allocation.editor()
```

Input Arguments

allocScenario — Allocation scenario

allocation scenario object

Allocation scenario from which to remove allocations, specified as a `systemcomposer.allocation.AllocationScenario` object.

sourceElement — Source element from which to delete allocation

element object

Source element from which to delete allocation, 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 from which to delete allocation

element object

Target element from which to delete allocation, 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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1 .	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`getAllocation` | `getAllocatedFrom` | `allocate` | `getAllocatedTo` | `destroy` | `getScenario` | `createAllocationSet`

Topics

“Create and Manage Allocations”

Introduced in R2020b

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"> “Author Software Architectures” “Simulate and Deploy Software Architectures”

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"> “Implement Behaviors for Architecture Model Simulation” “Create Software Architecture from Component”
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 the Software Architecture of a Throttle Position Control System”
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”

See Also

`systemcomposer.createModel` | `createArchitectureModel` | `increaseExecutionOrder`

Topics

“Modeling the Software Architecture of a Throttle Position Control System”

“Simulate and Deploy Software Architectures”

“Author Software Architectures”

Introduced in R2021b

systemcomposer.analysis.deleteInstance

Delete architecture instance

Syntax

```
systemcomposer.analysis.deleteInstance(architectureInstance)
```

Description

`systemcomposer.analysis.deleteInstance(architectureInstance)` deletes an existing instance.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `instance` refers to the element instance on which the iteration is being performed.

Examples

Delete Architecture Instance

Create a profile for latency characteristics.

```
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');
```

Instantiate all stereotypes in a profile.

```
model = systemcomposer.createModel('archModel', true);
instance = instantiate(model.Architecture, 'LatencyProfile', 'NewInstance');
```

Delete the architecture instance.

```
systemcomposer.analysis.deleteInstance(instance);
```

Input Arguments

architectureInstance — Architecture instance

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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an <code>.MAT</code> 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”

See Also

`instantiate` | `systemcomposer.analysis.Instance` | `loadInstance` | `save` | `refresh` | `update`

Topics

“Write Analysis Function”

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation', true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation', ...
    'Source_Model_Allocation', 'Target_Model_Allocation');

% Get the default allocation scenario
defaultScenario = allocSet.getScenario('Scenario 1');

% Create a new allocation scenario
newScenario = allocSet.createScenario('Scenario 2');

% Delete the default allocation scenario
allocSet.deleteScenario('Scenario 1');

% Save the allocation set
allocSet.save;

% 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 to be deleted

character vector | string

Name of allocation scenario to be deleted, 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 is a directed relationship from an element in one model to an element 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.	"Allocate Architectures in Tire Pressure Monitoring System"
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	"Create and Manage Allocations"
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

getScenario | createScenario | synchronizeChanges | load | closeAll | find | close

Topics

"Create and Manage Allocations"

Introduced in R2020b

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

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 named 'NewView'.

```
model.openViews
```

Create a subgroup.

```
group = view.Root.createSubGroup('MyGroup');
```

Delete the subgroup.

```
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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

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”

Introduced in R2021a

deleteView

Package: `systemcomposer.arch`

Delete architecture view

Syntax

```
deleteView(model,viewName)
```

Description

`deleteView(model,viewName)` deletes the view `viewName`, if it exists, in the specified model `model`.

Examples

Create and Delete 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
```

Delete the view and see that it has been deleted.

```
model.deleteView('NewView')
```

Input Arguments

model — Model

model object

Model, specified as a `systemcomposer.arch.Model` object.

viewName — Name of view

character vector | string

Name of view, specified as a character vector or string.

Example: 'NewView'

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 architecture design.	You can use different types of views to represent the system: <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”

Term	Definition	Application	More Information
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

`systemcomposer.view.View` | `openViews` | `getView` | `createView` | `systemcomposer.view.ElementGroup`

Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

Introduced in R2021a

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 named '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 | value type object | data interface object | data element object | physical domain object | physical interface object | physical element object | property object | view object | element group 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.ValueType`
- `systemcomposer.interface.DataInterface`
- `systemcomposer.interface.DataElement`
- `systemcomposer.interface.PhysicalDomain`
- `systemcomposer.interface.PhysicalInterface`

- `systemcomposer.interface.PhysicalElement`
- `systemcomposer.profile.Property`
- `systemcomposer.view.View`
- `systemcomposer.view.ElementGroup`

See Also

`Component` | `Variant Component` | `removeElement` | `removeElement` | `removeInterface` | `deleteView` | `deleteSubGroup` | `deleteInstance` | `removeProfile` | `removeProperty` | `removeStereotype` | `removeStereotype`

Introduced in R2019a

destroy

Package: systemcomposer.allocation

Remove allocation scenario or allocation

Syntax

```
destroy(allocScenario)
destroy(allocation)
```

Description

`destroy(allocScenario)` removes and destroys the existing allocation scenario `allocScenario` in an allocation set.

`destroy(allocation)` removes and destroys the existing allocation `allocation` in an allocation scenario.

Examples

Destroy Allocation Scenario

```
% 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 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    'Source_Model_Allocation','Target_Model_Allocation');

% Get the default allocation scenario
defaultScenario = allocSet.getScenario('Scenario 1');

% Destroy an allocation scenario in an allocation set
defaultScenario.destroy

% Save the allocation set
allocSet.save

% Open the allocation editor
systemcomposer.allocation.editor()
```

Input Arguments

allocScenario — Allocation scenario

allocation scenario object

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

allocation — Allocation

allocation object

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

More About

Definitions

Term	Definition	Application	More Information
allocation	An allocation is a directed relationship from an element in one model to an element 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.	"Allocate Architectures in Tire Pressure Monitoring System"
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	"Create and Manage Allocations"
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

`allocate` | `deallocate` | `createScenario` | `deleteScenario` | `getScenario` | `createAllocationSet`

Topics

"Create and Manage Allocations"

Introduced in R2020b

systemcomposer.allocation.editor

Open allocation editor

Syntax

```
systemcomposer.allocation.editor()
```

Description

systemcomposer.allocation.editor() opens the allocation editor.

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    'Source_Model_Allocation','Target_Model_Allocation');

% Save the allocation set
allocSet.save;

% Open the allocation editor
systemcomposer.allocation.editor()
```

More About

Definitions

Term	Definition	Application	More Information
allocation	An allocation is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	“Create and Manage Allocations”

Term	Definition	Application	More Information
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

`createAllocationSet` | `systemcomposer.allocation.AllocationSet`

Topics

"Create and Manage Allocations"

Introduced in R2020b

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.

```
profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');  
profile.save;  
systemcomposer.profile.editor(profile)
```

Input Arguments

profile – Profile

profile object

Profile to select, specified as a `systemcomposer.profile.Profile` object.

Example: `systemcomposer.profile.editor(profile)`

profileName – Name of profile

character vector | string

Name of profile to select, 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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

systemcomposer.profile.Profile | loadProfile | open | load | find | save | closeAll | createProfile

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

systemcomposer.exportModel

Export model information as MATLAB tables

Syntax

```
[exportedSet] = 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. 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”.

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, and requirementLinks, 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]
           domain: 'System'
```

Input Arguments

modelName — Name of model to be exported

character vector | string

Name of model to be exported, 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, and requirementLinks, with a domain field returned as 'System' for architecture models, and 'Software' for software architecture models.

Data Types: struct

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"

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 Interface Editor to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	<p>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 Manager on an architecture model or through custom views. Assign test cases to requirements using the Test Manager for verification and validation.</p>	“Link and Trace Requirements”
requirement set	A requirement set is a collection of requirements. You can structure the requirements hierarchically and link them to components or ports.	<p>Use the Requirements Editor to edit and refine requirements in a requirement set. Requirement sets are stored in <code>.slreqx</code> files. You can create a new requirement set and author requirements using Simulink Requirements™, or import requirements from supported third-party tools.</p>	“Manage Requirements”

Term	Definition	Application	More Information
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 <code>.slmx</code> files.	<ul style="list-style-type: none"> “Create an Architecture Model with Interfaces and Requirement Links” “Update Reference Requirement Links from Imported File” on page 1-606
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. The Interface Editor is accessible in System Composer test harness models to enable behavior testing and implementation-independent interface testing.	<ul style="list-style-type: none"> “Verify and Validate Requirements Using Test Harnesses on Components” “Create a Test Harness” (Simulink Test)

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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”

See Also

`importModel`

Topics

“Import and Export Architecture Models”

Introduced in R2019a

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 `architectureModelName` and saves it in the current directory.

`systemcomposer.extractArchitectureFromSimulink(model,name,Name,Value)` exports the Simulink model `model` to an architecture model `architectureModelName` 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 any, are also preserved.

Convert Simulink Model to System Composer Architecture

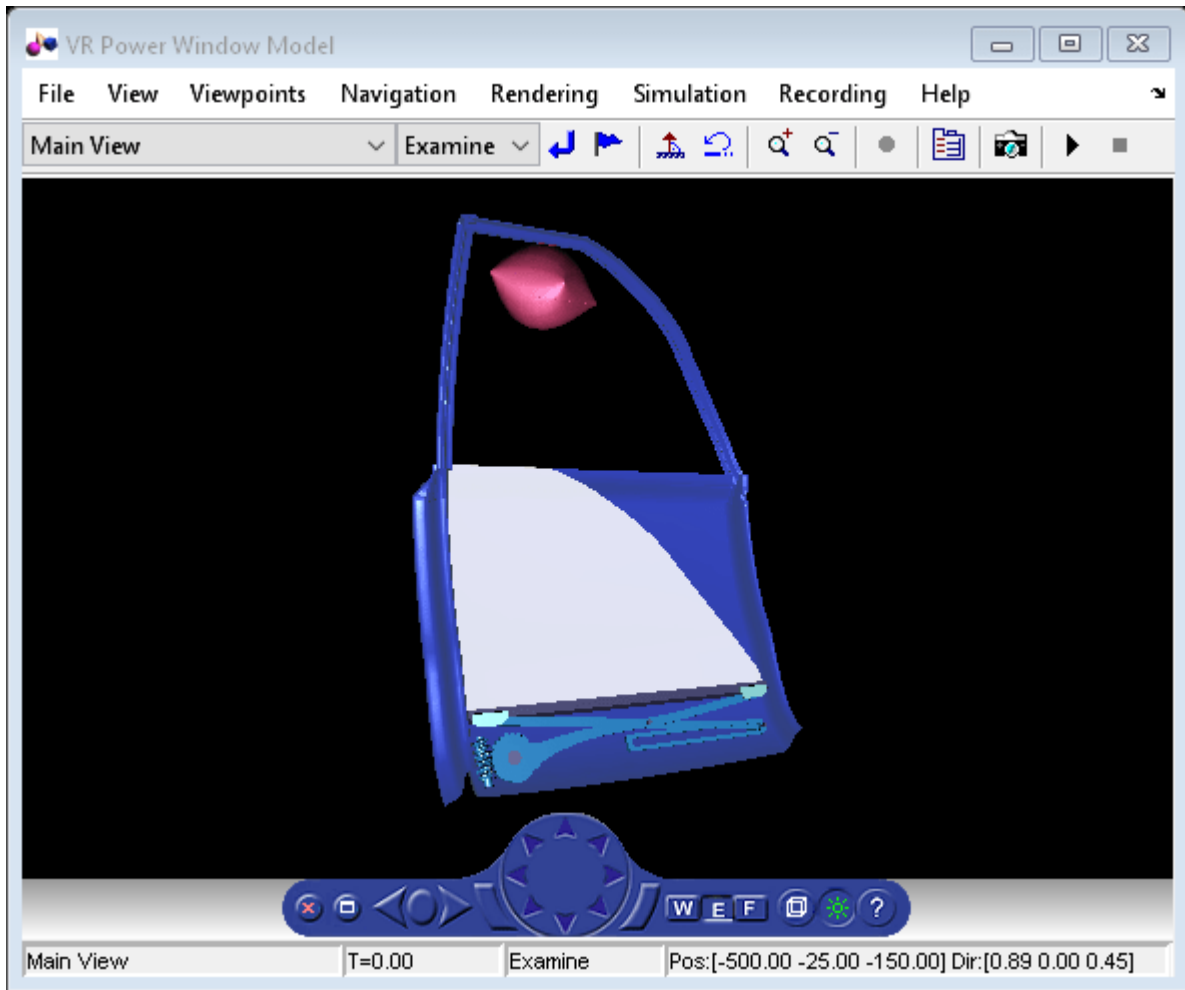
System Composer converts structural constructs in a Simulink model to equivalent architecture model constructs:

- Subsystems to components
- Variant subsystems to variant components
- Bus objects to interfaces
- Referenced models to reference components

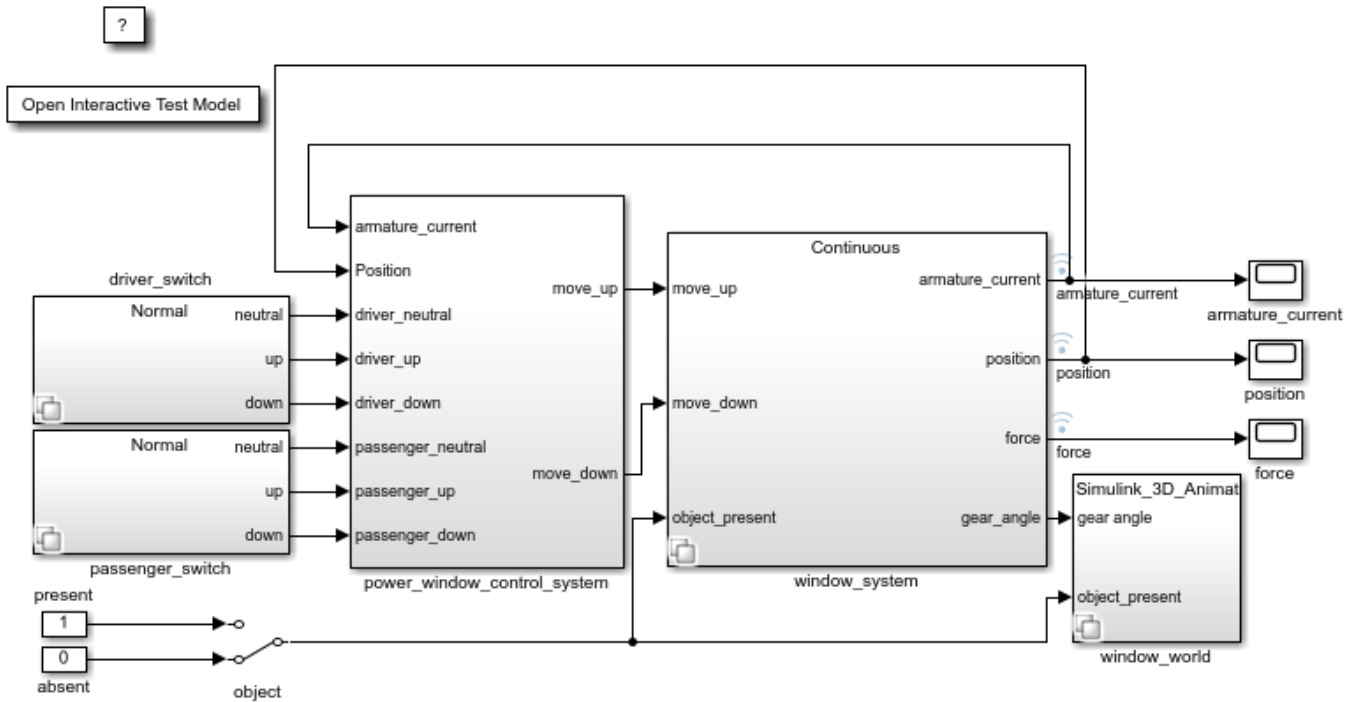
Open the Model

Open the Simulink model of the VR Power Window Model.

```
slxPowerWindowStart
```



```
open_system('slexPowerWindowExample');
```

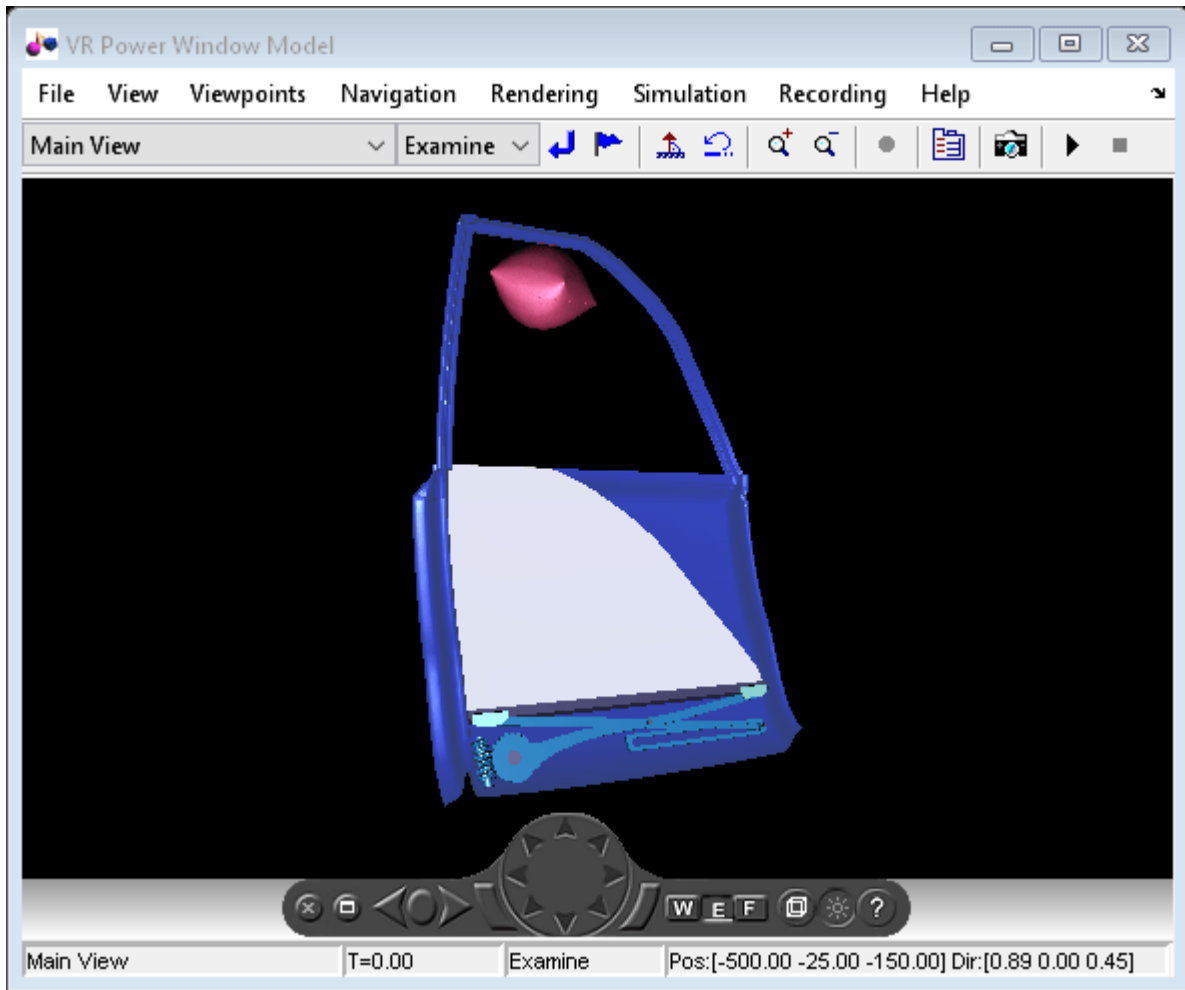


Copyright 2013-2016 The MathWorks, Inc.

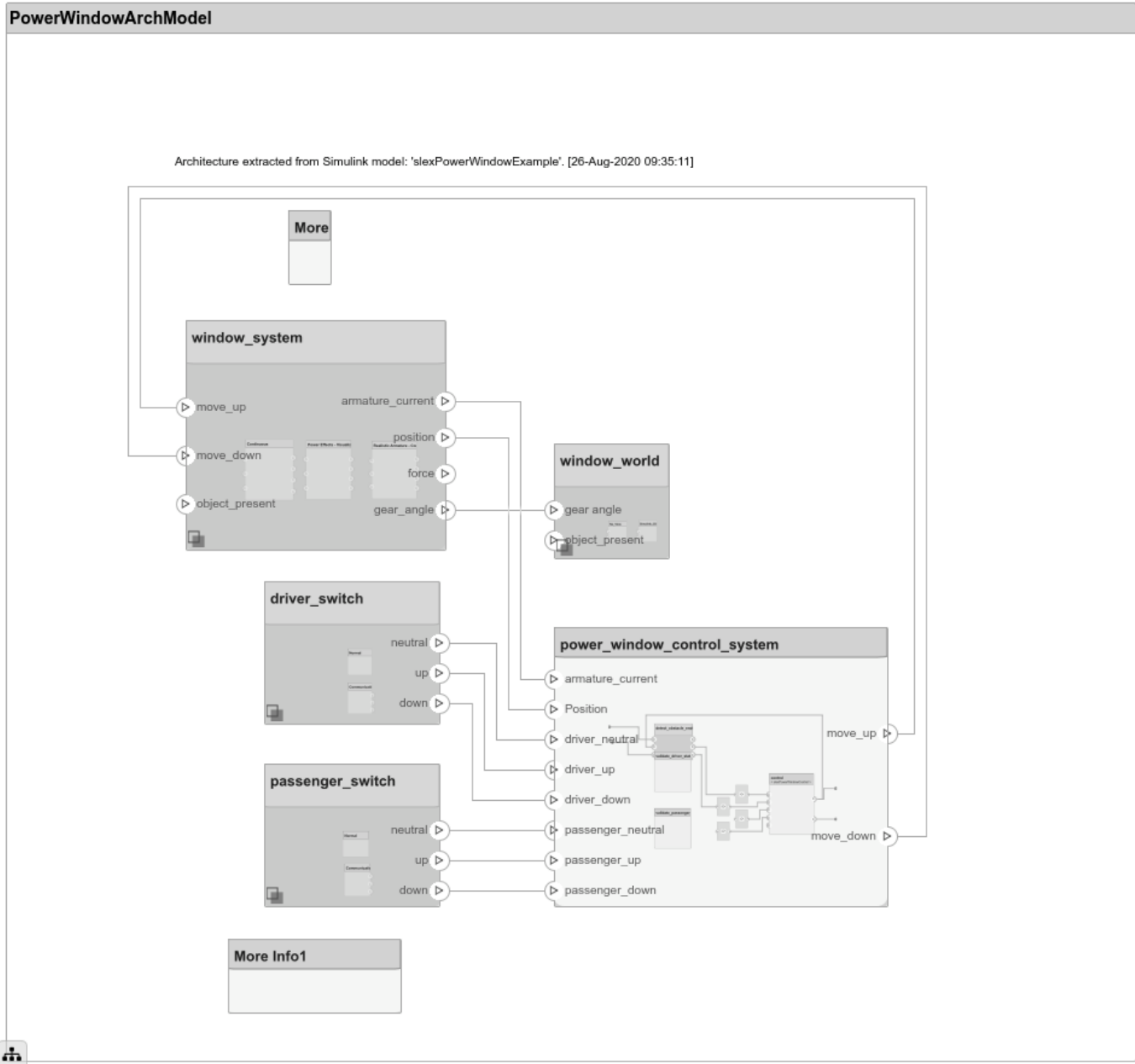
Export the Model

Extract an architecture model from the original model.

```
systemcomposer.extractArchitectureFromSimulink('slexPowerWindowExample','PowerWindowArchModel');
```



```
Simulink.BlockDiagram.arrangeSystem('PowerWindowArchModel');  
systemcomposer.openModel('PowerWindowArchModel');
```



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: 'slexPowerWindowExample'

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: 'PowerWindowArchModel'

Data Types: char | string

Name-Value Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:

```
systemcomposer.extractArchitectureFromSimulink('slexPowerWindowExample', 'PowerWindowArchModel', 'AutoArrange', false, 'ShowProgress', true)
```

AutoArrange — Option to auto-arrange architecture model

true or 1 (default) | false or 0

Option to auto-arrange architecture model, specified as a logical 1 (true) to auto-arrange or 0 (false) to not auto-arrange.

Example:

```
systemcomposer.extractArchitectureFromSimulink('slexPowerWindowExample', 'PowerWindowArchModel', 'AutoArrange', false)
```

Data Types: logical

ShowProgress — Option to show progress bar

false or 0 (default) | true or 1

Option to show progress bar, specified as a logical 0 (false) to not show a progress bar or 1 (true) to show a progress bar. This option is useful for larger models.

Example:

```
systemcomposer.extractArchitectureFromSimulink('slexPowerWindowExample', 'PowerWindowArchModel', '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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

`inlineComponent` | `createSimulinkBehavior` | `createStateflowChartBehavior` | `createSubsystemBehavior` | `extractArchitectureFromSimulink` | `linkToModel` | `isReference` | Reference Component

Topics

- "Extract Architecture from Simulink Model"
- "Describe Component Behavior Using Simulink"
- "Decompose and Reuse Components"
- "Describe Component Behavior Using Stateflow Charts"
- "Create Simulink Behavior Using Simulink Subsystem"

Introduced in R2019a

find

Package: systemcomposer.arch

Find architecture model elements using query

Syntax

```
[paths] = find(object,constraint,Name,Value)
[paths, elements] = find(____)
[elements] = find(____)
[paths] = find(object,constraint,rootArch,Name,Value)
```

Description

`[paths] = find(object,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(object,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 architecture elements that have a stereotype based on the specified sub-constraint.

```
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]  
    Position: [15 15 65 76]  
    Model: [1x1 systemcomposer.arch.Model]  
    SimulinkHandle: 2.0033  
    SimulinkModelHandle: 0.0033  
        UUID: '34e2d62d-4417-4012-a33e-d4e9ac5cbc38'  
    ExternalUID: ''
```

Clean Up

Uncomment to remove the model and the profile.

```
% m.close('force')
% systemcomposer.profile.Profile.closeAll
```

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 Architecture 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

object – Model

model object

Model, specified as a `systemcomposer.arch.Model` object to query using constraint.

constraint – Query

query constraint object

Query, specified as a `systemcomposer.query.Constraint` object representing specific conditions. A constraint can contain a sub-constraint 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.
HasPort	A component has a port that satisfies the given sub-constraint.
HasInterface	A port has an interface that satisfies the given sub-constraint.
HasInterfaceElement	An interface has an interface element that satisfies the given sub-constraint.
HasStereotype	An architecture element has a stereotype that satisfies the given sub-constraint.
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 comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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 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 or not, specified as 0 (false) to not search for referenced architectures or 1 (true) to search for referenced architectures.

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 'Component' to find components to satisfy the query, 'Port' to find ports to satisfy the query, or 'Connector' to find connectors to satisfy the query.

Example: `find(model,constraint,'ElementType','Port')`

Data Types: char

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

systemcomposer.query.Constraint | createView | lookup

Topics

“Create Architectural Views Programmatically”

Introduced in R2019a

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

`addStereotype` | `removeStereotype` | `getStereotype` | `getDefaultStereotype` | `setDefaultStereotype`

Topics

“Define Profiles and Stereotypes”
 “Use Stereotypes and Profiles”

“Modeling System Architecture of Small UAV”

Introduced in R2019a

systemcomposer.profile.Profile.find

Find profile by name

Syntax

```
profile = systemcomposer.profile.Profile.find()
profile = systemcomposer.profile.Profile.find(name)
```

Description

`profile = systemcomposer.profile.Profile.find()` finds all open profiles.

`profile = systemcomposer.profile.Profile.find(name)` finds a profile by the specified name, name.

Examples

Find Profile

Create a profile for latency characteristics.

```
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');
```

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

name — Name of profile

character vector | string

Name of profile to find, specified as a character vector or string.

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none">• “Define Profiles and Stereotypes”• “Use Stereotypes and Profiles”

See Also

systemcomposer.profile.Profile | open | editor | save | close | closeAll | load | createProfile

Topics

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

Introduced in R2019a

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 Default Scenario

```
% 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 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    'Source_Model_Allocation','Target_Model_Allocation');

% Find the default allocation scenario
defaultScenario = systemcomposer.allocation.AllocationSet.find('Scenario 1');

% Save the allocation set
allocSet.save;

% Open the allocation editor
systemcomposer.allocation.editor()
```

Input Arguments

name — Name of scenario to be found

character vector | string

Name of scenario to be found, specified as a character vector or string.

Example: 'Scenario 1'

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 is a directed relationship from an element in one model to an element 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.	"Allocate Architectures in Tire Pressure Monitoring System"
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1 .	"Create and Manage Allocations"
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

save | load | close | closeAll | synchronizeChanges | getScenario | createScenario | deleteScenario

Topics

"Create and Manage Allocations"

Introduced in R2020b

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 Choice

Create a model, get the root architecture, create one variant component, add two choices for the variant component, set the active choice, and find 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]
    Position: [15 15 65 65]
    Model: [1x1 systemcomposer.arch.Model]
  SimulinkHandle: 85.0006
  SimulinkModelHandle: 78.0002
    UUID: '23b62204-f0e2-48a2-8bd6-4689f003def4'
  ExternalUID: ''
```

Input Arguments

variantComponent — Variant component

variant component object

Variant component, specified as a `systemcomposer.arch.VariantComponent` object with multiple choices.

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 Condition” on page 1-531

See Also

`addChoice` | `getChoices` | `setActiveChoice` | Variant Component

Topics

“Create Variants”

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% Get allocated from source component allocated to target component
sourceElement = defaultScenario.getAllocatedFrom(targetComp);

% Save the allocation set
allocSet.save

% Open the allocation editor
systemcomposer.allocation.editor()
```

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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called <code>Scenario 1</code> .	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`getAllocatedTo` | `allocate` | `deallocate`

Topics

“Create and Manage Allocations”

Introduced in R2020b

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% Get allocated to target component allocated from source component
targetElement = defaultScenario.getAllocatedTo(sourceComp);

% Save the allocation set
allocSet.save

% 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.

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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called <code>Scenario 1</code> .	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`allocate` | `getAllocatedFrom` | `deallocate`

Topics

“Create and Manage Allocations”

Introduced in R2020b

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);
mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    'Source_Model_Allocation','Target_Model_Allocation');

% Get the default allocation scenario
defaultScenario = allocSet.getScenario('Scenario 1');

% Allocate components between models
allocation = defaultScenario.allocate('Source_Component','Target_Component');

% Get the allocation between the source component and the target component
allocation = defaultScenario.getAllocation('Source_Component','Target_Component');
```

Input Arguments

allocScenario — Allocation scenario

allocation scenario object

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

sourceElement — Source element for allocation

element object

Source element for allocation, 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 for allocation

element object

Target element for allocation, 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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

getAllocatedTo | getAllocatedFrom | deallocate | allocate

Topics

“Create and Manage Allocations”

Introduced in R2020b

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 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);
choices(1)
```

ans =

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]
Position: [15 15 65 65]
Model: [1x1 systemcomposer.arch.Model]
SimulinkHandle: 99.0010
SimulinkModelHandle: 94.0002
UUID: '533d7f63-41e2-40fd-afe8-d081729849f0'
ExternalUID: ''
```

Input Arguments

variantComponent — Variant component

variant component object

Variant component, specified as a `systemcomposer.arch.VariantComponent` object with multiple choices.

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 Condition" on page 1-531

See Also

`addChoice` | `getActiveChoice` | `setActiveChoice` | Variant Component

Topics

"Create Variants"

Introduced in R2019a

getCondition

Package: systemcomposer.arch

Return variant control on choice within variant component

Syntax

```
expression = getCondition(variantComponent,choice)
```

Description

`expression = getCondition(variantComponent,choice)` returns the variant control on the choice within the variant component.

Examples

Get Condition

Create a model, get the root architecture, create on variant component, add two choices for the variant component, set the active variant choice, set a condition, and get the condition.

```
model = systemcomposer.createModel('archModel',true);
arch = get(model,'Architecture');
mode = 1;
variant = addVariantComponent(arch,'Component1');
compList = addChoice(variant,{'Choice1','Choice2'});
setActiveChoice(variant,compList(2));
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 with multiple choices.

choice — Choice in variant component

component object

Choice in variant component whose control string is returned by this function, specified by 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 Condition" on page 1-531

See Also

makeVariant | setActiveChoice | setCondition | addVariantComponent | Variant Component

Topics

"Create Variants"

Introduced in R2019a

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.

```
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', 'AppliesTo', 'Connector');
connLatency.addProperty('secure', 'Type', 'boolean');
connLatency.addProperty('linkDistance', 'Type', 'double');

nodeLatency = profile.addStereotype('NodeLatency', 'Parent', ...
'LatencyProfile.LatencyBase', 'AppliesTo', 'Component');
nodeLatency.addProperty('resources', 'Type', 'double', 'DefaultValue', '1');

portLatency = profile.addStereotype('PortLatency', 'Parent', ...
'LatencyProfile.LatencyBase', 'AppliesTo', 'Port');
portLatency.addProperty('queueDepth', 'Type', 'double');
portLatency.addProperty('dummy', 'Type', 'int32');
```

Set the default component stereotype.

```
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'

Element type, specified as 'Component', 'Port', 'Connector', or 'Interface'.

Data Types: char | string

stereotype — Stereotype of parent element

stereotype object

Stereotype of parent element, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

[applyStereotype](#) | [removeStereotype](#) | [setDefaultElementStereotype](#)

Topics

“Define Profiles and Stereotypes”

Introduced in R2021b

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: 'default'
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]

```

Uncomment the following code and run to clean up the artifacts created by this example.

```
% systemcomposer.profile.Profile.closeAll
```

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none">• “Set Properties”• “Add Properties with Stereotypes”• “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none">• “Define Profiles and Stereotypes”• “Use Stereotypes and Profiles”

See Also

`createProfile` | `setDefaultStereotype` | `addStereotype` | `getStereotype` | `removeStereotype`

Topics

“Create a Profile and Add Stereotypes”

Introduced in R2019a

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 connection.

Examples

Selected Element on Destination Port Connection

Get the selected element on the destination port for a connection.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
rootArch = get(arch,'Architecture'); % Get architecture

newComponent = addComponent(rootArch,'Component1'); % Add component
outPortComp = addPort(newComponent.Architecture,...
'testSig','out'); % Create out-port on component
outPortArch = addPort(rootArch,'testSig','out'); % Create out-port on architecture
compSrcPort = getPort(newComponent,'testSig'); % Extract component port object
archDestPort = getPort(rootArch,'testSig'); % Extract architecture port object

interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
interface.addElement('x'); % Create interface element
archDestPort.setInterface(interface); % Set interface on architecture port

conns = connect(compSrcPort,archDestPort,'DestinationElement','x'); % Connect ports
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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. <p>System Composer models are stored as SLX files.</p>	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"

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 Interface Editor to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addPort` | `getPort` | `addComponent` | `addElement` | `addInterface` | `setInterface` | `connect` | `getSourceElement` | `Component`

Topics

“Specify a Source Element or Destination Element for Ports on a Connection”

Introduced in R2020b

getElement

Package: systemcomposer.interface

Get object for data element

Syntax

```
element = getElement(interface,elementName)
```

Description

`element = getElement(interface,elementName)` gets the object for the element `elementName` in the interface specified by `interface`.

Examples

Get Object for Named Data Element

Add a data interface named 'newInterface' to the interface dictionary of the model. Add a data element named '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 named 'newInterface' to the interface dictionary of the model. Add a physical element named 'newElement' with 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

Interface to add the element to, specified as a `systemcomposer.interface.DataInterface` or `systemcomposer.interface.PhysicalInterface` object.

elementName – Name of element

character vector | string

Name of element to identify, specified as a character vector or string.

Data Types: `char` | `string`

Output Arguments

element – Element

data element object | physical element object

Element, returned as a `systemcomposer.interface.DataElement` or `systemcomposer.interface.PhysicalElement` 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> “Manage Interfaces with Data Dictionaries” “Reference Data Dictionaries”

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 Interface Editor to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`addElement` | `removeElement` | `createDictionary` | `getInterfaceNames` | `getInterface` | `linkDictionary` | `getSourceElement` | `getDestinationElement` | `unlinkDictionary`

Topics

- "Specify Physical Interfaces on the Ports"
- "Create Interfaces"
- "Manage Interfaces with Data Dictionaries"

Introduced in R2019a

getEvaluatedPropertyValue

Package: systemcomposer.arch

Get evaluated value of property from component

Syntax

```
value = getEvaluatedPropertyValue(element,property)
```

Description

`value = getEvaluatedPropertyValue(element,property)` obtains the evaluated value of a property specified on the architecture element.

Examples

Get Evaluated 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');
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 = getEvaluatedPropertyValue(comp,'LatencyProfile.electricalComponent.latency')
```

```
value =
```

```
10
```

Input 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, 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.interface.DataInterface, systemcomposer.ValueType, or systemcomposer.interface.PhysicalInterface 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • "Set Properties" • "Add Properties with Stereotypes" • "Set Properties for Analysis"
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • "Define Profiles and Stereotypes" • "Use Stereotypes and Profiles"

See Also

setProperty | getProperty | getStereotypeProperties | getPropertyValue

Topics

"Write Analysis Function"

Introduced in R2019a

getInterface

Package: systemcomposer.interface

Get object for named interface in interface dictionary

Syntax

```
interface = getInterface(dictionary,name)
```

Description

`interface = getInterface(dictionary,name)` gets the object for a named interface in the interface dictionary.

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'
  ExternalUUID: ''
```

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'
ExternalUID: ''

```

Input Arguments

dictionary – Data dictionary

dictionary object

Data dictionary, specified as a `systemcomposer.interface.Dictionary` object. This is the data dictionary attached to the model. It could be the local dictionary of the model or an external data dictionary.

name – Name of interface

character vector | string

Name of interface, specified as a character vector or string.

Data Types: `char` | `string`

Output Arguments

interface – Interface

data interface object | physical interface object | value type object

Interface, returned as a `systemcomposer.interface.DataInterface`, `systemcomposer.interface.PhysicalInterface`, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> “Manage Interfaces with Data Dictionaries” “Reference Data Dictionaries”

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 Interface Editor to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`addElement` | `getInterfaceNames` | `removeElement` | `addInterface` | `addValueType` | `addPhysicalInterface` | `Adapter`

Topics

"Specify Physical Interfaces on the Ports"
 "Create Interfaces"
 "Manage Interfaces with Data Dictionaries"

Introduced in R2019a

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 attached to the model, specified as a `systemcomposer.interface.Dictionary` object for the local dictionary of the model or an external data dictionary.

Output Arguments

interfaceNames – Interface names

array of character vectors

Interface names, returned as an 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`addInterface` | `getInterface` | `removeInterface` | `addValueType` | `addPhysicalInterface` | `Adapter`

Topics

"Specify Physical Interfaces on the Ports"
 "Create Interfaces"
 "Manage Interfaces with Data Dictionaries"

Introduced in R2019a

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 to find, specified as a character vector or string.

Example: 'testSig'

Data Types: char | string

Output Arguments

port — Port of component

component port

Port of component, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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”

See Also

`createModel` | `addPort` | `addComponent` | `connect` | `Component`

Introduced in R2019a

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>'.
'<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, 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 — Model element

architecture object | component object | port object | connector object | physical connector object | data interface object | value type object | physical interface object

Model 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.interface.DataInterface`, `systemcomposer.ValueType`, or `systemcomposer.interface.PhysicalInterface` 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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”

See Also

setProperty | removeProperty | addProperty | getStereotypeProperties

Topics

“Set Properties for Analysis”

Introduced in R2019a

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 architecture 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 — Model element

architecture object | component object | port object | connector object | physical connector object | data interface object | value type object | physical interface object

Model 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.interface.DataInterface, systemcomposer.ValueType, or systemcomposer.interface.PhysicalInterface 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`setProperty` | `getStereotypeProperties` | `getProperty` | `getEvaluatedPropertyValue`

Topics

"Write Analysis Function"

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    'Source_Model_Allocation','Target_Model_Allocation');

% Get the default allocation scenario
defaultScenario = allocSet.getScenario('Scenario 1');

% Save the allocation set
allocSet.save

% 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 — 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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`createScenario` | `deleteScenario` | `close` | `load` | `save` | `synchronizeChanges` | `find` | `closeAll`

Topics

“Create and Manage Allocations”

Introduced in R2020b

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 connection.

Examples

Selected Element on Source Port Connection

Get the selected element on the source port for a connection.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
rootArch = get(arch,'Architecture'); % Get architecture

newComponent = addComponent(rootArch,'Component1'); % Add component
inPortComp = addPort(newComponent.Architecture,...
'testSig','in'); % Create in-port on component
inPortArch = addPort(rootArch,'testSig','in'); % Create in-port on architecture
compDestPort = getPort(newComponent,'testSig'); % Extract component port object
archSrcPort = getPort(rootArch,'testSig'); % Extract architecture port object

interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
interface.addElement('x'); % Create interface element
archSrcPort.setInterface(interface); % Set interface on architecture port

conns = connect(archSrcPort,compDestPort,'SourceElement','x'); % Connect ports
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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"

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 Interface Editor to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addPort` | `getPort` | `addComponent` | `addElement` | `addInterface` | `setInterface` | `connect` | `getDestinationElement` | `Component`

Topics

“Specify a Source Element or Destination Element for Ports on a Connection”

Introduced in R2020b

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: 'default'
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]

Uncomment the following code and run to clean up the artifacts created by this example.

```
% systemcomposer.profile.Profile.closeAll
```

Input Arguments

profile – Profile

profile object

Profile with stereotypes, specified as a `systemcomposer.profile.Profile` object.

name – Name of stereotype

character vector | string

Name of stereotype to find, specified as a character vector or string.

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none">• “Set Properties”• “Add Properties with Stereotypes”• “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none">• “Define Profiles and Stereotypes”• “Use Stereotypes and Profiles”

See Also

[addStereotype](#) | [removeStereotype](#) | [getDefaultStereotype](#) | [setDefaultStereotype](#)

Topics

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

Introduced in R2019a

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

Model element architecture, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, `systemcomposer.arch.PhysicalConnector`, `systemcomposer.interface.DataInterface`, `systemcomposer.ValueType`, or `systemcomposer.interface.PhysicalInterface` 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. <p>System Composer models are stored as SLX files.</p>	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

[setProperty](#) | [getProperty](#) | [getEvaluatedPropertyValue](#) | [getPropertyValue](#)

Topics

"Write Analysis Function"

Introduced in R2019a

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 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');  
  
model.applyProfile('LatencyProfile');
```

Apply the stereotype to the component, open the profile editor, and get the stereotypes on the component.

```
comp.applyStereotype('LatencyProfile.LatencyBase');
```

```
systemcomposer.profile.editor(profile)
```

```
stereotypes = getStereotypes(comp)
```

```
stereotypes =
```

```
    1x1 cell array
```

```
{'LatencyProfile.LatencyBase'}
```

Input 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, 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.interface.DataInterface`, `systemcomposer.ValueType`, or `systemcomposer.interface.PhysicalInterface` 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. <p>System Composer models are stored as SLX files.</p>	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	<p>Local interfaces on a System Composer model can be saved in an interface data dictionary using the Interface Editor.</p> <p>Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.</p>	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”

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 Interface Editor to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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”

See Also

`applyStereotype` | `removeStereotype` | `batchApplyStereotype` | `getStereotypeProperties`

Topics

“Use Stereotypes and Profiles”

Introduced in R2019a

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

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 named 'NewView'.

```
model.openViews
```

Create a subgroup.

```
group = view.Root.createSubGroup('MyGroup');
```

Get the subgroup.

```
getGroup = view.Root.getSubGroup('MyGroup')
```

```
getGroup =
```

```
    ElementGroup with properties:
```

```
        Name: 'MyGroup'
        UUID: '46eaaed7-3ba0-418e-bc65-1ef8bce3087b'
        Elements: []
        SubGroups: [0x0 systemcomposer.view.ElementGroup]
```

Input Arguments

elementGroup — Element group

element group object

Element group for a 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.

Data Types: char | string

Output Arguments**subGroup — Subgroup**

element group object

Subgroup, returned as a `systemcomposer.view.ElementGroup` 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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. <i>Functional views</i> focus on what the system must do to operate. <i>Physical views</i> show how the system is constructed and configured. <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"> “Create Architecture Views Interactively” “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

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”

Introduced in R2021a

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.

This function is part of the systemcomposer.analysis.Instance class that you can use to analyze the model iteratively, element by element. instance refers to the element instance on which the iteration is being performed.

Examples

Get Mass Property Value

Load the Small 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 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.

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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”

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 an .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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

setValue | hasValue | systemcomposer.analysis.Instance

Topics

“Write Analysis Function”

Introduced in R2019a

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 named '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

component object | variant component object | architecture port object | component port object | connector object | physical connector object

Architecture model element, specified as one of these objects:

- `systemcomposer.arch.Component`
- `systemcomposer.arch.VariantComponent`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.Connector`
- `systemcomposer.arch.PhysicalConnector`

See Also

Component | Variant Component | lookup

Introduced in R2019a

getView

Package: systemcomposer.arch

Find architecture view

Syntax

```
view = getView(model,viewName)
```

Description

`view = getView(model,viewName)` finds the view `view` in the architecture model `model` with view name `viewName`.

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
```

Delete the view and see that it has been deleted.

```
foundView = model.getView('NewView')
```

```
foundView =
```

```
View with properties:
```

```
        Name: 'NewView'
        Root: [1x1 systemcomposer.view.ElementGroup]
        Model: [1x1 systemcomposer.arch.Model]
        UUID: 'ff912f2c-5cdd-4dda-9125-fb6b819b3f7a'
        Select: []
        GroupBy: {}
        Color: '#0072bd'
        Description: ''
        IncludeReferenceModels: 1
```

Input Arguments

model — Model

model object

Model, specified as a `systemcomposer.arch.Model` object.

viewName — Name of view

character vector | string

Name of view, specified as a character vector or string.

Example: 'NewView'

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

systemcomposer.view.View | createView | deleteView | openViews | systemcomposer.view.ElementGroup

Topics

“Create Architecture Views Interactively”
 “Create Architectural Views Programmatically”

Introduced in R2021a

HasInterface

Package: systemcomposer.query

Create query to select architecture elements with interface on port based on specified sub-constraint

Syntax

```
query = HasInterface(sub-constraint)
```

Description

`query = HasInterface(sub-constraint)` creates a query `query` that the `find` function and the `createView` function use to select architecture elements with an interface that satisfies the given sub-constraint `sub-constraint`.

Examples

Construct Query to Select All Port Interfaces

Select all of the port interfaces in an architecture model with matching criteria.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Open the Simulink project file.

```
scKeylessEntrySystem
```

Open the model.

```
m = systemcomposer.openModel('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(m,constraint,'Recurse',true,'IncludeReferenceModels',true)
```

```
portInterfaces =
```

```
10x1 cell array
```

```
{'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

sub-constraint — Condition restricting the query

query constraint object

Condition restricting the query, specified as a `systemcomposer.query.Constraint` object.

Example: `contains(Property('Name'), 'KeyFOBPosition')`

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"

Term	Definition	Application	More Information
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `HasPort` | `HasInterfaceElement`

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

HasInterfaceElement

Package: systemcomposer.query

Create query to select architecture elements with interface element on interface based on specified sub-constraint

Syntax

```
query = HasInterfaceElement(sub-constraint)
```

Description

`query = HasInterfaceElement(sub-constraint)` creates a query `query` that the `find` function and the `createView` function use to select architecture elements with an interface element that satisfies the given sub-constraint `sub-constraint`.

Examples

Construct Query to Select All Interface Elements

Select all of the port interface elements in an architecture model with matching criteria.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Open the Simulink project file.

```
scExampleSmallUAV
```

Open the model.

```
m = systemcomposer.openModel('scExampleSmallUAVModel');
```

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(m,constraint,'Recurse',true,'IncludeReferenceModels',true)
```

```
elements =
```

```
4x1 cell array
```

```

{'scExampleSmallUAVModel/FlightComputer'      }
{'scExampleSmallUAVModel/FlightComputer/Main Board'}
{'scExampleSmallUAVModel/Payload'             }
{'scExampleSmallUAVModel/Payload/Payload'     }
```

Input Arguments

sub-constraint — Condition restricting the query

query constraint object

Condition restricting the query, specified as a `systemcomposer.query.Constraint` object.

Example: `contains(Property('Name'), 'c')`

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

createView | find | systemcomposer.query.Constraint | HasInterface | HasPort

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

HasPort

Package: `systemcomposer.query`

Create query to select architecture elements with port on component based on specified sub-constraint

Syntax

```
query = HasPort(sub-constraint)
```

Description

`query = HasPort(sub-constraint)` creates a query `query` that the `find` function and the `createView` function use to select architecture elements with a port that satisfies the given sub-constraint `sub-constraint`.

Examples

Construct Query to Select All Sensor Component Ports

Select all of the sensor component ports in an architecture model.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Open the Simulink project file.

```
scKeylessEntrySystem
```

Open the model.

```
m = systemcomposer.openModel('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(m,constraint,'Recurse',true,'IncludeReferenceModels',true)
```

```
sensorComp =
```

```
1x1 cell array
```

```
{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller'}
```

Input Arguments

sub-constraint — Condition restricting the query

query constraint object

Condition restricting the query, specified as a `systemcomposer.query.Constraint` object.

Example: `contains(Property('Name'), 'Sensor')`

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. <i>Functional views</i> focus on what the system must do to operate. <i>Physical views</i> show how the system is constructed and configured. <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"> “Create Architecture Views Interactively” “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `HasInterface` | `HasInterfaceElement`

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

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, 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 — Architecture model element

component object | variant component object | component port object | architecture port object | connector object | physical connector object | data interface object | value type object | physical interface object

Architecture model element, specified as one of these objects:

- `systemcomposer.arch.Component`
- `systemcomposer.arch.VariantComponent`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.arch.PhysicalConnector`
- `systemcomposer.interface.DataInterface`
- `systemcomposer.ValueType`
- `systemcomposer.interface.PhysicalInterface`

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	<p>Local interfaces on a System Composer model can be saved in an interface data dictionary using the Interface Editor.</p> <p>Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.</p>	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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.	<p>Data interfaces are decomposed into data elements:</p> <ul style="list-style-type: none"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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”

See Also

`addProperty` | `removeProperty` | `hasStereotype`

Topics

“Use Stereotypes and Profiles”

Introduced in R2021a

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, 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');
model.applyProfile('LatencyProfile');
```

Apply the stereotype to the component. Open the profile editor, then find if the stereotypes are applied on the component.

```
comp.applyStereotype('LatencyProfile.LatencyBase');
systemcomposer.profile.editor()
profile.save
result = hasStereotype(comp, 'LatencyProfile.LatencyBase')
result =
    logical
```

1

Input Arguments

element — Architecture model element

component object | variant component object | component port object | architecture port object | connector object | physical connector object | data interface object | value type object | physical interface object

Architecture model element, specified as one of these objects:

- `systemcomposer.arch.Component`
- `systemcomposer.arch.VariantComponent`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.arch.PhysicalConnector`
- `systemcomposer.interface.DataInterface`
- `systemcomposer.ValueType`
- `systemcomposer.interface.PhysicalInterface`

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	<p>Local interfaces on a System Composer model can be saved in an interface data dictionary using the Interface Editor.</p> <p>Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.</p>	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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.	<p>Data interfaces are decomposed into data elements:</p> <ul style="list-style-type: none"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

[removeStereotype](#) | [applyStereotype](#) | [hasProperty](#) | [getStereotypes](#)

Topics

"Use Stereotypes and Profiles"

Introduced in R2021a

HasStereotype

Package: systemcomposer.query

Create query to select architecture elements with stereotype based on specified sub-constraint

Syntax

```
query = HasStereotype(sub-constraint)
```

Description

`query = HasStereotype(sub-constraint)` creates a query `query` that the `find` function and the `createView` function use to select architecture elements with a stereotype that satisfies the given sub-constraint `sub-constraint`.

Examples

Construct Query to Select All Hardware Components

Select all of the hardware components in an architecture model.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Open the Simulink project file.

```
scKeylessEntrySystem
```

Open the model.

```
m = systemcomposer.openModel('KeylessEntryArchitecture');
```

Create a query for all the hardware components and run the query, displaying one of them.

```
constraint = HasStereotype(IsStereotypeDerivedFrom('AutoProfile.HardwareComponent'));  
hwComp = find(m,constraint, 'Recurse',true, 'IncludeReferenceModels',true);  
hwComp(16)
```

```
ans =
```

```
1x1 cell array
```

```
{'KeylessEntryArchitecture/F0B Locator System/Center Receiver/PWM'}
```

Input Arguments

sub-constraint — Condition restricting the query

query constraint object

Condition restricting the query, specified as a `systemcomposer.query.Constraint` object.

Example: `IsStereotypeDerivedFrom('AutoProfile.HardwareComponent')`

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. <i>Functional views</i> focus on what the system must do to operate. <i>Physical views</i> show how the system is constructed and configured. <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"> “Create Architecture Views Interactively” “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

`createView` | `find` | `systemcomposer.query.Constraint` | `IsStereotypeDerivedFrom`

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

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`.

This function is part of the `systemcomposer.analysis.Instance` class that you can use to analyze the model iteratively, element by element. `instance` refers to the element instance on which the iteration is being performed.

Examples

Query Whether Instance Has Property

Use the `hasValue` function to 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 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
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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an .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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

setValue | getValue | systemcomposer.analysis.Instance

Topics

“Write Analysis Function”

Introduced in R2019a

systemcomposer.importModel

Import model information from MATLAB tables

Syntax

```
archModel = systemcomposer.importModel(modelName,components,ports,  
connections,portInterfaces,requirementLinks)  
archModel = systemcomposer.importModel(modelName,importStruct)  
[archModel,idMappingTable,importLog,errorLog] = systemcomposer.importModel(  
___ )
```

Description

`archModel = systemcomposer.importModel(modelName,components,ports,connections,portInterfaces,requirementLinks)` creates a new architecture model based on MATLAB tables that specify components, ports, connections, port interfaces, and requirement links. 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 a Basic Architecture”. To import `requirementLinks`, you need a Simulink Requirements 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 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)` API:

- 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
- Saves referenced models
- Saves the architecture model

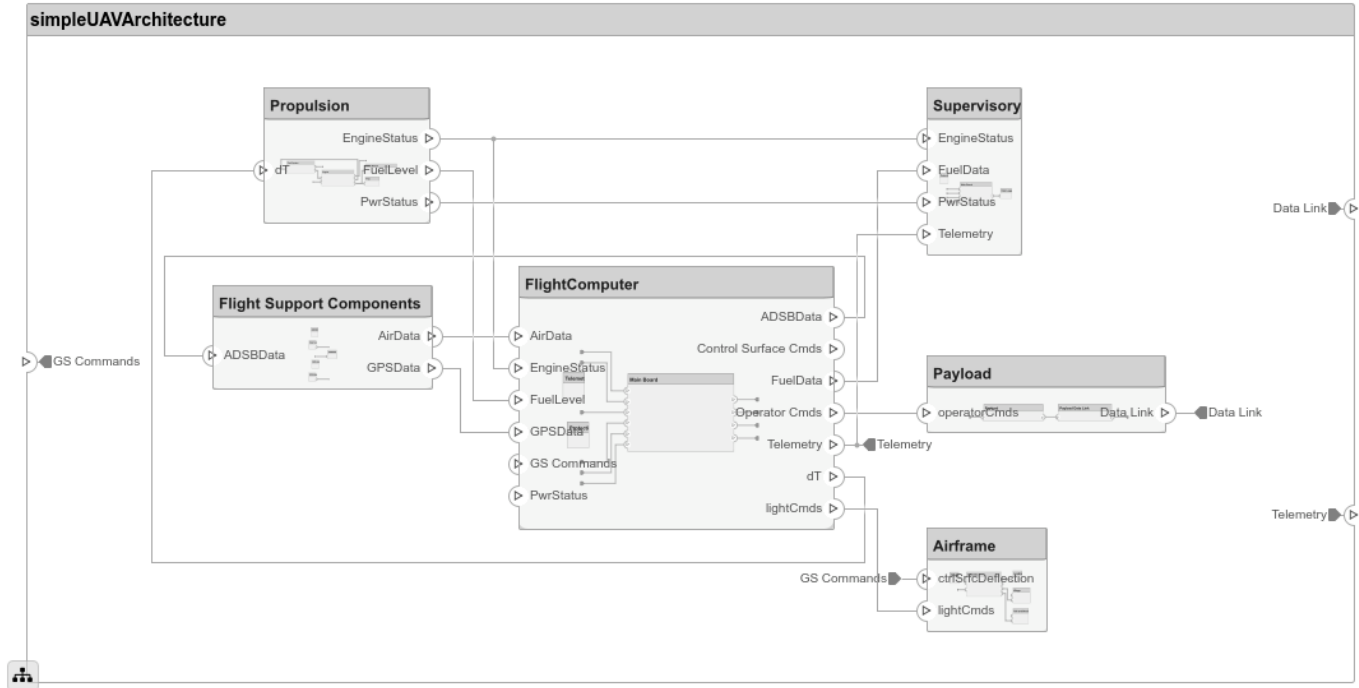
```
% Instantiate adapter class to read from Excel.
modelName = 'simpleUAVArchitecture';

% 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 the above structure to Excel file.
SaveToExcel('ExportedUAVModel',exportedSet);
```

Input Arguments

modelName — Name of model

character vector | string

Name of model to be created, specified as a character vector or string.

Example: 'importedModel'

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 1-606. Optional columns include: DestinationArtifact, SourceArtifact, ReferencedReqID, Keywords, CreatedOn, CreatedBy, ModifiedOn, ModifiedBy, and Revision. A Simulink Requirements license is required to import the requirementLinks table to a System Composer architecture model.

Data Types: table

importStruct — Model tables

structure

Model tables, specified as a structure containing the tables components, ports, connections, portInterfaces, and requirementLinks, 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 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	<p>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 Manager on an architecture model or through custom views. Assign test cases to requirements using the Test Manager for verification and validation.</p>	“Link and Trace Requirements”
requirement set	A requirement set is a collection of requirements. You can structure the requirements hierarchically and link them to components or ports.	<p>Use the Requirements Editor to edit and refine requirements in a requirement set. Requirement sets are stored in <code>.slreqx</code> files. You can create a new requirement set and author requirements using Simulink Requirements, or import requirements from supported third-party tools.</p>	“Manage Requirements”

Term	Definition	Application	More Information
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 <code>.slmx</code> files.	<ul style="list-style-type: none"> “Create an Architecture Model with Interfaces and Requirement Links” “Update Reference Requirement Links from Imported File” on page 1-606
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. The Interface Editor is accessible in System Composer test harness models to enable behavior testing and implementation-independent interface testing.	<ul style="list-style-type: none"> “Verify and Validate Requirements Using Test Harnesses on Components” “Create a Test Harness” (Simulink Test)

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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”

See Also

`exportModel` | `systemcomposer.updateLinksToReferenceRequirements` | Component | Variant Component | Reference Component

Topics

“Import and Export Architecture Models”

Introduced in R2019a

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"> “Author Software Architectures” “Simulate and Deploy Software Architectures”

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"> • “Implement Behaviors for Architecture Model Simulation” • “Create Software Architecture from Component”
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 the Software Architecture of a Throttle Position Control System”
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”

See Also

systemcomposer.createModel | createArchitectureModel | decreaseExecutionOrder

Topics

“Modeling the Software Architecture of a Throttle Position Control System”

“Simulate and Deploy Software Architectures”

“Author Software Architectures”

Introduced in R2021b

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 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 `electricComp` to the same model `Robot.slx` so it uses the architecture of `robotComp` and references it.

```
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);
```

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);
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • “Describe Component Behavior Using Simulink” • “Create Reference Architecture”
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"> • “Create Simulink Behavior Using Simulink Subsystem” • “Create a Simulink Subsystem Component”

Term	Definition	Application	More Information
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"> “Implement Behaviors for Architecture Model Simulation” “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> “Describe System Behavior Using Sequence Diagrams” “Use Sequence Diagrams with Architecture Models”

See Also

createSimulinkBehavior | createSubsystemBehavior | createArchitectureModel | createStateflowChartBehavior | extractArchitectureFromSimulink | isReference | Reference Component

Topics

“Describe Component Behavior Using Simulink”
 “Decompose and Reuse Components”
 “Describe Component Behavior Using Stateflow Charts”
 “Create Simulink Behavior Using Simulink Subsystem”
 “Simulate and Deploy Software Architectures”

Introduced in R2019a

instantiate

Package: systemcomposer.arch

Create analysis instance from specification

Syntax

```
instance = instantiate(model,properties,name)
instance = instantiate(model,profile,name)
instance = instantiate( ____,Name,Value)
```

Description

`instance = instantiate(model,properties,name)` creates an instance `instance` named `name` of a model architecture `model` with properties `properties` for analysis.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `instance` refers to the element instance on which the iteration is being performed.

`instance = instantiate(model,profile,name)` creates an instance `instance` named `name` of a model architecture `model` 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 Stereotype

Instantiate all properties of a stereotype that will be applied to specific elements during instantiation.

Create a profile for latency characteristics.

```
profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');

% Add base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency','Type','double');
latencybase.addProperty('dataRate','Type','double','DefaultValue','10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency','Parent',...
'LatencyProfile.LatencyBase');
connLatency.addProperty('secure','Type','boolean');
connLatency.addProperty('linkDistance','Type','double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency','Parent',...
'LatencyProfile.LatencyBase');
nodeLatency.addProperty('resources','Type','double','DefaultValue','1');

% Add port stereotype with properties
portLatency = profile.addStereotype('PortLatency','Parent',...
'LatencyProfile.LatencyBase');
```

```
portLatency.addProperty('queueDepth','Type','double');
portLatency.addProperty('dummy','Type','int32');
```

Instantiate all properties of a stereotype.

```
model = systemcomposer.createModel('archModel',true); % Create new model
model.applyProfile('LatencyProfile'); % Apply profile to model

% 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 stereotype
instance = instantiate(model.Architecture,properties,'NewInstance')
```

Instantiate Specific Properties of Stereotype

Instantiate specific properties of a stereotype that will be applied to specific elements during instantiation.

Create a profile for latency characteristics.

```
profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');

% Add base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency','Type','double');
latencybase.addProperty('dataRate','Type','double','DefaultValue','10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency','Parent',...
    'LatencyProfile.LatencyBase');
connLatency.addProperty('secure','Type','boolean');
connLatency.addProperty('linkDistance','Type','double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency','Parent',...
    'LatencyProfile.LatencyBase');
nodeLatency.addProperty('resources','Type','double','DefaultValue','1');

% Add port stereotype with properties
portLatency = profile.addStereotype('PortLatency','Parent',...
    'LatencyProfile.LatencyBase');
portLatency.addProperty('queueDepth','Type','double');
portLatency.addProperty('dummy','Type','int32');
```

Instantiate specific properties of a stereotype.

```
model = systemcomposer.createModel('archModel',true); % Create new model
model.applyProfile('LatencyProfile'); % Apply profile to model

% Specify some properties of stereotypes
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 stereotype
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.

```

profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');

% Add base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency','Type','double');
latencybase.addProperty('dataRate','Type','double','DefaultValue','10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency','Parent',...
    'LatencyProfile.LatencyBase');
connLatency.addProperty('secure','Type','boolean');
connLatency.addProperty('linkDistance','Type','double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency','Parent',...
    'LatencyProfile.LatencyBase');
nodeLatency.addProperty('resources','Type','double','DefaultValue','1');

% Add port stereotype with properties
portLatency = profile.addStereotype('PortLatency','Parent',...
    'LatencyProfile.LatencyBase');
portLatency.addProperty('queueDepth','Type','double');
portLatency.addProperty('dummy','Type','int32');

```

Instantiate all stereotypes in a profile.

```

model = systemcomposer.createModel('archModel',true); % Create new model
model.applyProfile('LatencyProfile'); % Apply profile to model

% Instantiate all stereotypes in 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 base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency','Type','double');
latencybase.addProperty('dataRate','Type','double','DefaultValue','10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency','Parent',...
'LatencyProfileC.LatencyBase');
connLatency.addProperty('secure','Type','boolean','DefaultValue','true');
connLatency.addProperty('linkDistance','Type','double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency','Parent',...
'LatencyProfileC.LatencyBase');
nodeLatency.addProperty('resources','Type','double','DefaultValue','1');

% Add 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); % Create new model
arch = model.Architecture;

model.applyProfile('LatencyProfileC'); % Apply profile to model

% Create components, ports, and connections
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
planningPorts = addPort(components(2).Architecture,{'SensorData','MotionCommand'},{'in','out'});
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
c_sensorData = connect(arch,components(1),components(2));
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));

% Clean up 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

```

Clean Up

Uncomment the following code and run to clean up the artifacts created by this example.

```

% bdclose('archModel')
% systemcomposer.profile.Profile.closeAll

```

Input Arguments

model — Model architecture

architecture object

Model architecture from which instance is generated, specified as a `systemcomposer.arch.Architecture` object.

Example: `model.Architecture`

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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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 1 (`true`) to normalize or 0 (`false`) to do nothing.

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 – Analysis direction

`'TopDown' | 'PreOrder' | 'PostOrder' | 'BottomUp'`

Analysis direction, specified as a character vector or string.

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

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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"> • “Analyze an Architecture Model with an Analysis Function” • “Analyze Architecture”

Term	Definition	Application	More Information
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an .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”

See Also

systemcomposer.analysis.Instance | deleteInstance | loadInstance | save | update | iterate

Topics

“Write Analysis Function”

Introduced in R2019a

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.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an <code>.MAT</code> 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

systemcomposer.analysis.Instance | isComponent | isConnector | isPort

Topics

"Write Analysis Function"

Introduced in R2019a

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.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an <code>.MAT</code> 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

isArchitecture | isConnector | isPort | systemcomposer.analysis.Instance

Topics

"Write Analysis Function"

Introduced in R2019a

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.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an <code>.MAT</code> 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

systemcomposer.analysis.Instance | isArchitecture | isComponent | isPort

Topics

"Write Analysis Function"

Introduced in R2019a

IsInRange

Package: systemcomposer.query

Create query to select range of property values

Syntax

```
query = IsInRange(propertyName,beginRangeValue,endRangeValue)
```

Description

query = IsInRange(propertyName,beginRangeValue,endRangeValue) creates a query that the find function and the createView function use to select a range of values from beginRangeValue to endRangeValue for a specified property name propertyName.

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.

```
scKeylessEntrySystem
```

Open the model.

```
m = systemcomposer.openModel('KeylessEntryArchitecture');
```

Create a query to find values from 10 ms to 40 ms in the 'Latency' property.

```
constraint = IsInRange(PropertyValue('AutoProfile.BaseComponent.Latency'),...
Value(10,'ms'),Value(40,'ms'));
latency = find(m,constraint,'Recurse',true,'IncludeReferenceModels',true)
```

```
latency =
```

```
5×1 cell array
```

```
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Actuator'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Actuator' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Actuator' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Actuator' }
{'KeylessEntryArchitecture/Sound System/Dashboard Speaker' }
```

Input Arguments

propertyName — Property name

character vector

Property name for model element, specified as a character vector as fully qualified name '`<profile>.<stereotype>.<property>`' or any property on the designated object.

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

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

`createView` | `find` | `systemcomposer.query.Constraint`

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

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.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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"> • “Analyze an Architecture Model with an Analysis Function” • “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an .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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

isArchitecture | isComponent | isConnector | systemcomposer.analysis.Instance

Topics

"Write Analysis Function"

Introduced in R2019a

isProtected

Package: systemcomposer.arch

Find if component referenced 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.

The referenced model on the component is not protected.

```
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 Simulink model to the component 'newComponent'
linkToModel(newComponent, 'newReference.slxp');

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • "Describe Component Behavior Using Simulink" • "Create Reference Architecture"

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"> • “Create Simulink Behavior Using Simulink Subsystem” • “Create a Simulink Subsystem Component”
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"> • “Implement Behaviors for Architecture Model Simulation” • “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> • “Describe System Behavior Using Sequence Diagrams” • “Use Sequence Diagrams with Architecture Models”

See Also

`inlineComponent` | `createSimulinkBehavior` | `createArchitectureModel` | `createStateflowChartBehavior` | `extractArchitectureFromSimulink` | `linkToModel` | `isReference` | Reference Component

Topics

“Describe Component Behavior Using Simulink”
 “Decompose and Reuse Components”
 “Describe Component Behavior Using Stateflow Charts”
 “Create Simulink Behavior Using Simulink Subsystem”
 “Simulate and Deploy Software Architectures”

Introduced in R2021b

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.

The component is not a reference.

```
model = systemcomposer.createModel('archModel',true);
rootArch = get(model,'Architecture');
newComponent = addComponent(rootArch,'newComponent');
flag = isReference(newComponent)
```

```
flag =
    logical
     0
```

The component is a reference.

```
model = systemcomposer.createModel('archModel');
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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • "Describe Component Behavior Using Simulink" • "Create Reference Architecture"

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"> “Create Simulink Behavior Using Simulink Subsystem” “Create a Simulink Subsystem Component”
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"> “Implement Behaviors for Architecture Model Simulation” “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> “Describe System Behavior Using Sequence Diagrams” “Use Sequence Diagrams with Architecture Models”

See Also

`inlineComponent` | `createSubsystemBehavior` | `createSimulinkBehavior` | `createArchitectureModel` | `createStateflowChartBehavior` | `extractArchitectureFromSimulink` | `linkToModel` | Reference Component

Topics

“Describe Component Behavior Using Simulink”
 “Decompose and Reuse Components”
 “Describe Component Behavior Using Stateflow Charts”
 “Create Simulink Behavior Using Simulink Subsystem”
 “Simulate and Deploy Software Architectures”

Introduced in R2019a

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` function and the `createView` function use to select a stereotype from the qualified name `name`.

Examples

Construct Query to Select All Hardware Components

Select all of the hardware components in an architecture model.

Import the package that contains all of the System Composer queries.

```
import systemcomposer.query.*
```

Open the Simulink project file.

```
scKeylessEntrySystem
```

Open the model.

```
m = systemcomposer.openModel('KeylessEntryArchitecture');
```

Create a query for all the hardware components and run the query, displaying one of them.

```
constraint = HasStereotype(IsStereotypeDerivedFrom('AutoProfile.HardwareComponent'));  
hwComp = find(m,constraint,'Recurse',true,'IncludeReferenceModels',true);  
hwComp(16)
```

```
ans =
```

```
1x1 cell array
```

```
{'KeylessEntryArchitecture/FOB Locator System/Center Receiver/PWM'}
```

Input Arguments

name — Stereotype name

character vector

Stereotype name, specified as a character vector in the form '`<profile>.<stereotype>`'.

Example: 'AutoProfile.BaseComponent'

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	"Display Component Hierarchy and Architecture Hierarchy Using Views"

See Also

createView | find | systemcomposer.query.Constraint | HasStereotype

Topics

"Create Architectural Views Programmatically"

Introduced in R2019b

iterate

Package: systemcomposer.arch

Iterate over model elements

Syntax

```
iterate(architecture,iterType,iterFunction)
iterate( ____,Name,Value)
iterate( ____,additionalArgs)
```

Description

`iterate(architecture,iterType,iterFunction)` iterates over components in the architecture `architecture` in the order specified by `iterType` and invokes the function specified by the function handle `iterFunction` on each component.

`iterate(____,Name,Value)` iterates over components in the architecture, with additional options.

`iterate(____,additionalArgs)` passes all trailing arguments, specified as `additionalArgs`, as arguments to `iterFunction`.

Examples

Compute Battery Capacity

Open the example “Battery Sizing and Automotive Electrical System Analysis”.

```
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,objcomputeBatterySizing);
```

Input Arguments

architecture – 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'.

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 comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `iterate(archModel, 'Topdown', @computeLoad, 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' does not apply to a `systemcomposer.analysis.ArchitectureInstance` object. The architecture model is flattened.

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.

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 has to be either 'TopDown' or 'BottomUp'. If any other option is specified, iteration defaults to 'TopDown'.

'FollowConnectivity' does not apply to a `systemcomposer.analysis.ArchitectureInstance` object.

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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"> • “Analyze an Architecture Model with an Analysis Function” • “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”

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 an .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"

See Also

instantiate | lookup | systemcomposer.analysis.Instance

Topics

"Analyze Architecture"

Introduced in R2019a

linkDictionary

Package: systemcomposer.arch

Link data dictionary to architecture model

Syntax

```
linkDictionary(modelObject,dictionaryFile)
```

Description

`linkDictionary(modelObject,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

modelObject – Architecture model

model object

Architecture model from which the dictionary link is to be added, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"
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.	"Create an Architecture Model with Interfaces and Requirement Links"

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

[createDictionary](#) | [saveToDictionary](#) | [unlinkDictionary](#) | [openDictionary](#) | [addReference](#) | [removeReference](#)

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2019a

linkToModel

Package: systemcomposer.arch

Link component to a model

Syntax

```
modelHandle = linkToModel(component,modelName)
modelHandle = linkToModel(component,modelFileName)
```

Description

`modelHandle = linkToModel(component,modelName)` links from the component to a model.

`modelHandle = linkToModel(component,modelFileName)` links from the component to a model 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 named 'electricComp' uses the architecture of the component named '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 'electricComp' to the same model 'Robot.slx' so it uses the architecture of 'robotComp' and references it.

```
linkToModel(comp(1),'Robot');
```

Input Arguments

component — Component

component object

Component with no sub-components, specified as a `systemcomposer.arch.Component` object.

modelName — Model name

character vector | string

Model name for an existing model that defines the architecture or behavior of the component, specified as a character vector or string. Models of the same name prioritize protected models.

Example: 'Robot'

Data Types: char | string

modelName — Model file name

character vector | string

Model file name for an existing model 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

numeric value

Handle to linked model, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • “Describe Component Behavior Using Simulink” • “Create Reference Architecture”
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"> • “Create Simulink Behavior Using Simulink Subsystem” • “Create a Simulink Subsystem Component”
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"> • “Implement Behaviors for Architecture Model Simulation” • “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> • “Describe System Behavior Using Sequence Diagrams” • “Use Sequence Diagrams with Architecture Models”

See Also

createSubsystemBehavior | inlineComponent | createSimulinkBehavior | createArchitectureModel | createStateflowChartBehavior | extractArchitectureFromSimulink | isReference | Reference Component

Topics

“Describe Component Behavior Using Simulink”
 “Decompose and Reuse Components”
 “Describe Component Behavior Using Stateflow Charts”
 “Create Simulink Behavior Using Simulink Subsystem”
 “Simulate and Deploy Software Architectures”

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% Save the allocation set
allocSet.save;

% Close the allocation set
allocSet.close;

% 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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`createAllocationSet` | `open` | `closeAll`

Topics

“Create and Manage Allocations”

Introduced in R2020b

systemcomposer.profile.Profile.load

Load profile from file

Syntax

```
profile = systemcomposer.profile.Profile.load(fileName)
```

Description

`profile = systemcomposer.profile.Profile.load(fileName)` 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

fileName — File name for profile

character vector | string

File name for profile, specified as a character vector or string. Profile must be available on the MATLAB path.

Example: 'ProfileName.xml'

Example: 'LatencyProfile'

Data Types: char | string

Output Arguments

profile — Loaded profile

profile object

Loaded 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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

systemcomposer.profile.Profile | open | editor | save | find | closeAll | close | createProfile

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

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.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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.

```
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');
```

Instantiate all stereotypes in a 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

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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”

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 an .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"

See Also

instantiate | systemcomposer.analysis.Instance | deleteInstance | save | refresh | update

Topics

"Write Analysis Function"

Introduced in R2019a

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.

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 architecture model

character vector | string

Name of architecture model, specified as a character vector or string. Architecture model must exist on the MATLAB path.

Example: 'new_arch'

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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">• <i>Component ports</i> are interaction points on the component to other components.• <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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”

See Also

save | open | `systemcomposer.createModel`

Topics

“Create an Architecture Model”

Introduced in R2019a

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, open the profile editor, and 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()
```

```
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 an `.xml` extension.

Example: 'new_profile'

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

`applyProfile` | `createProfile` | `editor` | `systemcomposer.profile.Profile`

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

lookup

Package: systemcomposer.arch

Search for architecture element

Syntax

```
element = lookup(object,Name,Value)
instance = lookup(object,Name,Value)
```

Description

`element = lookup(object,Name,Value)` finds an architecture element based on its universal unique identifier (UUID) or full path.

`instance = lookup(object,Name,Value)` finds an architecture element instance based on its universal unique identifier (UUID) or full path.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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]
    Position: [349 74 469 174]
    Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 7.0043
    SimulinkModelHandle: 0.0043
    UUID: 'cfd62628-d365-47e4-8492-62cfeaa8dc15'
    ExternalUID: ''
```

Input Arguments

object — Architecture model object

model object

Architecture model object to look up, specified as a `systemcomposer.arch.Model` object.

Name-Value Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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

Model element, returned as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.arch.PhysicalConnector` object.

instance — Element instance

architecture instance | component instance | port instance | connector instance

Element instance, returned as a `systemcomposer.analysis.ArchitectureInstance`, `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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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"> • “Analyze an Architecture Model with an Analysis Function” • “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”

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 an .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"

See Also

find | createView | getQualifiedName | systemcomposer.view.ElementGroup | systemcomposer.analysis.Instance | iterate | instantiate

Topics

"Analyze Architecture"

"Create Architectural Views Programmatically"

Introduced in R2019a

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 and available variant choice components.

`[variantComp,choices] = makeVariant(component,Name,Value)` converts the component `component` to a variant choice component with additional options and returns the parent variant component and available variant choice components.

Examples

Make Variant Component

Create two components with two ports each.

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 components.

```
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]
    Position: [15 13 65 81]
    Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 69.0001
SimulinkModelHandle: 1.2207e-04
    UUID: 'ee705b8f-b383-4230-a1a2-3c69fb081cc5'
    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]
    Position: [50 20 100 76]
    Model: [1x1 systemcomposer.arch.Model]
    SimulinkHandle: 62.0001
SimulinkModelHandle: 1.2207e-04
    UUID: '5ad838ca-f993-4349-aac9-2efca6d2066e'
    ExternalUID: ''

```

Open the system and arrange it. Save the model.

```

open(arch)
Simulink.BlockDiagram.arrangeSystem('archModel');
save(arch)

```

Input Arguments

component — Architecture component

component object

Architecture component to be converted to variant choice component, specified as a `systemcomposer.arch.Component` object.

Name-Value Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

```

Example: [variantComp,choices] =
makeVariant(newComponent, 'Name', 'NewVariantComponent', 'Label', 'NewVariantChoice', 'Choices',
{'NewVariantChoiceA', 'NewVariantChoiceB', 'NewVariantChoiceC'}, 'ChoiceLabels',
{'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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Condition" on page 1-531

See Also

addChoice | getChoices | Variant Component | addVariantComponent

Topics

"Create Variants"

Introduced in R2019a

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`. If an empty cell array `{}` is passed into `groupBy`, all the groupings are removed.

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 named '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 to modify, 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 sub-constraint 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.
HasPort	A component has a port that satisfies the given sub-constraint.
HasInterface	A port has an interface that satisfies the given sub-constraint.
HasInterfaceElement	An interface has an interface element that satisfies the given sub-constraint.
HasStereotype	An architecture element has a stereotype that satisfies the given sub-constraint.
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

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.

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

systemcomposer.view.View | createView | getView | deleteView | openViews | runQuery | removeQuery | systemcomposer.view.ElementGroup

Topics

“Create Architecture Views Interactively”
 “Create Architectural Views Programmatically”

Introduced in R2021a

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.

```
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');
```

Open the profile in the Profile Editor.

```
open(profile)
```

Input Arguments

profile — Profile

profile object

Profile to open in Profile Editor, 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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

createProfile | find | editor | save | load | close | closeAll

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% Save the allocation set
allocSet.save;

% 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 is a directed relationship from an element in one model to an element 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.	"Allocate Architectures in Tire Pressure Monitoring System"
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1 .	"Create and Manage Allocations"
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

`createAllocationSet | load`

Topics

"Create and Manage Allocations"

Introduced in R2020b

open

Package: `systemcomposer.arch`

Open architecture model

Syntax

```
open(objModel)
```

Description

`open(objModel)` opens the specified model in System Composer.

`open` is a method for the class `systemcomposer.arch.Model`.

Examples

Create and Open Model

```
model = systemcomposer.createModel('modelName');  
open(model)
```

Input Arguments

objModel — Model to open in editor

model object

Model to open in editor, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

`createModel` | `openModel`

Topics

"Create an Architecture Model"

Introduced in R2019a

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.slidd');  
dictionary = systemcomposer.openDictionary('my_dictionary.slidd');
```

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 `.slidd` extension.

Example: `'my_dictionary.slidd'`

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"
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.	"Create an Architecture Model with Interfaces and Requirement Links"

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

[linkDictionary](#) | [saveToDictionary](#) | [unlinkDictionary](#) | [createDictionary](#) | [addReference](#) | [removeReference](#)

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2019a

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.

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 architecture model to open, specified as a character vector or string. The model must exist on the MATLAB path.

Example: 'new_arch'

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

[open](#) | [close](#)

Topics

"Create an Architecture Model"

Introduced in R2019a

openViews

Package: systemcomposer.arch

Open architecture views editor

Syntax

```
openViews(model)
```

Description

`openViews(model)` opens the architecture views editor for the specified model, `model`. If the model is already open, `openViews` will bring the views to the front.

The method `openViews` is from the class `systemcomposer.arch.Model`.

Examples

Open Views Editor

Create a view component with a context view. Open the views editor for a model.

```
scKeylessEntrySystem
model = systemcomposer.loadModel('KeylessEntryArchitecture');
fobSupplierView = model.createView('FOB Locator System Supplier Breakdown',...
    'Color','lightblue');

% Open the views editor and see the new view in light blue
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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 architecture design.	You can use different types of views to represent the system: <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"

Term	Definition	Application	More Information
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

systemcomposer.view.View | createView | getView | deleteView |
systemcomposer.view.ElementGroup

Topics

“Create Architecture Views Interactively”
“Create Architectural Views Programmatically”

Introduced in R2019b

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` function and the `createView` function 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.

```
scKeylessEntrySystem
```

Open the model.

```
m = systemcomposer.openModel('KeylessEntryArchitecture');
```

Create a query to find components that contain the character vector 'Sensor' in their 'Name' property and run the query, displaying the first.

```
constraint = contains(Property('Name'),'Sensor');  
sensors = find(m,constraint,'Recurse',true,'IncludeReferenceModels',true);  
sensors(1)
```

```
ans =
```

```
1x1 cell array
```

```
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Sensor'}
```

Input Arguments

name — Property name

character vector

Property name for model element, specified as a character vector 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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

createView | find | systemcomposer.query.Constraint | PropertyValue

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

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 function and the createView function 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.

```
scKeylessEntrySystem
```

Open the model.

```
m = systemcomposer.openModel('KeylessEntryArchitecture');
```

Create a query to find components that contain the character vector 'Sensor' in their 'Name' property and run the query.

```
constraint = PropertyValue('AutoProfile.BaseComponent.Latency')==30;
latency = find(m,constraint,'Recurse',true,'IncludeReferenceModels',true)
```

```
latency =
```

```
4x1 cell array
```

```
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Actuator'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Actuator' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Actuator' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Actuator' }
```

Input Arguments

name — Property name

character vector

Property name for model element, specified as a character vector 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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

[createView](#) | [find](#) | [systemcomposer.query.Constraint](#) | [Property](#)

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

refresh

Package: `systemcomposer.analysis`

Refresh architecture instance

Syntax

```
refresh(architectureInstance)
```

Description

`refresh(architectureInstance)` refreshes an architecture instance `architectureInstance` to mirror the changes in the specification model. The `refresh` method is part of the `systemcomposer.analysis.ArchitectureInstance` class.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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.

```
profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');  
  
latencybase = profile.addStereotype('LatencyBase');  
latencybase.addProperty('latency', 'Type', 'double');  
latencybase.addProperty('dataRate', 'Type', 'double', 'DefaultValue', '10');
```

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 'dataRate' on the architecture instance.

```
instance.refresh();  
value = instance.getValue('LatencyProfile.LatencyBase.dataRate')
```

value =

10

Input Arguments

architectureInstance — Architecture instance

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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an <code>.MAT</code> 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”

See Also

instantiate | systemcomposer.analysis.Instance | loadInstance | deleteInstance | update | save | lookup | iterate

Topics

“Write Analysis Function”

Introduced in R2019a

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

Compatibility Considerations**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”

Introduced in R2019b

removeElement

Package: systemcomposer.interface

Remove data element

Syntax

```
removeElement(interface,elementName)
```

Description

removeElement(interface,elementName) removes an element from an interface.

Examples

Remove Data Element from Data Interface

Add a data interface named 'newInterface' to the interface dictionary of the model. Add a data element named '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 named 'newInterface' to the interface dictionary of the model. Add a physical element named 'newElement' with type 'electrical.electrical' to the physical interface, then remove the physical element.

```
arch = systemcomposer.createModel('newModel',true);
interface = addPhysicalInterface(arch.InterfaceDictionary,'newInterface');
element = addElement(interface,'newElement','Type','electrical.electrical');
removeElement(interface,'newElement')
```

Input Arguments

interface — Interface

data interface object | physical interface object

Interface to remove element from specified as a systemcomposer.interface.DataInterface or systemcomposer.interface.PhysicalInterface object.

elementName — Name of element

character vector | string

Name of element to be removed, specified as a character vector or string.

Example: 'newElement'

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`addElement` | `createDictionary` | `getElement` | `getInterfaceNames` | `getInterface` | `linkDictionary` | `getSourceElement` | `getDestinationElement` | `unlinkDictionary`

Topics

- "Specify Physical Interfaces on the Ports"
- "Create Interfaces"
- "Manage Interfaces with Data Dictionaries"

Introduced in R2019a

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 `removeElement` cannot be used when a selection query or grouping is defined on the view. To remove the query, run `removeQuery`.

Examples

Add Elements and Remove Elements from 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 named '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 from the view by path.

```
view.Root.removeElement('KeylessEntryArchitecture/Lighting System/Headlights')
```

Remove an element from 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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. <i>Functional views</i> focus on what the system must do to operate. <i>Physical views</i> show how the system is constructed and configured. <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"> "Create Architecture Views Interactively" "Create Architectural Views Programmatically"
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

openViews | createView | getView | deleteView | systemcomposer.view.ElementGroup | systemcomposer.view.View | addElement | getSubGroup | deleteSubGroup | createSubGroup

Topics

“Create Architecture Views Interactively”

“Create Architectural Views Programmatically”

Introduced in R2021a

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 named '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 attached to architecture model, specified as a `systemcomposer.interface.Dictionary` object.

name — Name of interface

character vector | string

Name of interface to be removed, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`addInterface` | `addValueType` | `addPhysicalInterface` | `getInterface` | `getInterfaceNames` | `Adapter`

Topics

"Specify Physical Interfaces on the Ports"
 "Create Interfaces"
 "Manage Interfaces with Data Dictionaries"

Introduced in R2019a

removeProfile

Package: systemcomposer.arch

Remove profile from model

Syntax

```
removeProfile(modelObject,profileName)
```

Description

removeProfile(modelObject,profileName) removes the profile from a model.

Examples

Remove Profile

Create a model.

```
model = systemcomposer.createModel('archModel',true);
```

Create a profile with a stereotype, open the profile editor, and 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

modelObject — 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none">• “Define Profiles and Stereotypes”• “Use Stereotypes and Profiles”

See Also

[applyProfile](#) | [createProfile](#)

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

removeProperty

Package: systemcomposer.profile

Remove property from stereotype

Syntax

```
removeProperty(stereotype,propertyName)
```

Description

`removeProperty(stereotype,propertyName)` removes a property from the stereotype.

Examples

Remove a 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 from which property is removed

stereotype object

Stereotype from which property is removed, 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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

[addProperty](#) | [setProperty](#) | [getProperty](#)

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

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 and 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 called '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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

systemcomposer.view.View | createView | getView | deleteView | openViews | runQuery | modifyQuery | systemcomposer.view.ElementGroup

Topics

- “Create Architecture Views Interactively”
- “Create Architectural Views Programmatically”

Introduced in R2021a

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 an interface named 'newInterface' to the local interface dictionary of the model. Save the local interface dictionary to a shared dictionary as an .sldd file.

```
% Create a new model and add an interface to its local dictionary
arch = systemcomposer.createModel('newModel',true);
addInterface(arch.InterfaceDictionary,'newInterface');
```

```
% Save interfaces from a local dictionary to a shared dictionary
saveToDictionary(arch,'TopDictionary')
```

```
% Open the shared dictionary
topDictionary = systemcomposer.openDictionary('TopDictionary.sldd');
```

Create a new dictionary and add it as a reference to the existing dictionary.

```
% Create a new dictionary
refDictionary = systemcomposer.createDictionary('ReferenceDictionary.sldd');
```

```
% Add the new dictionary as a reference
addReference(topDictionary,'ReferenceDictionary.sldd')
```

Remove the referenced dictionary.

```
% Remove the referenced dictionary
removeReference(topDictionary,'ReferenceDictionary.sldd')
```

Input Arguments

dictionary – Dictionary

dictionary object

Dictionary, specified as a systemcomposer.interface.Dictionary object.

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

saveToDictionary | createDictionary | openDictionary | linkDictionary | unlinkDictionary | addReference

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021a

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

Add a component stereotype to the profile and remove it.

```
profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');  
stereotype = addStereotype(profile, 'electricalComponent', 'AppliesTo', 'Component');  
profile.removeStereotype('electricalComponent')
```

Input Arguments

profile — Profile object

profile

Profile object, 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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

[getStereotype](#) | [addStereotype](#) | [getDefaultStereotype](#) | [setDefaultStereotype](#)

Topics

“Create a Profile and Add Stereotypes”

Introduced in R2019a

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, 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');  
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 — Model element

architecture object | component object | port object | connector object | physical connector object | data interface object | value type object | physical interface object

Model 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.interface.DataInterface`, `systemcomposer.ValueType`, or `systemcomposer.interface.PhysicalInterface` 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	<p>Local interfaces on a System Composer model can be saved in an interface data dictionary using the Interface Editor.</p> <p>Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.</p>	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

[applyStereotype](#) | [batchApplyStereotype](#) | [getStereotypes](#) | [getStereotypeProperties](#)

Topics

"Remove a Stereotype"

Introduced in R2019a

renameProfile

Package: systemcomposer.arch

Rename profile in model

Syntax

```
renameProfile(modelName,oldProfileName,newProfileName)
```

Description

`renameProfile(modelName,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.

Examples

Rename Profile

Create a model.

```
model = systemcomposer.createModel('archModel',true);
```

Create a profile with a stereotype, 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');
```

```
model.applyProfile('LatencyProfile');
```

Save the model and close the model.

```
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

modelName — Model architecture

model object | character vector | string

Model architecture, specified as a `systemcomposer.arch.Model` object or a character vector or string as the name of the model.

Example: 'MyModel'

Example: `archModel`

Data Types: `char` | `string`

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • "Set Properties" • "Add Properties with Stereotypes" • "Set Properties for Analysis"
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • "Define Profiles and Stereotypes" • "Use Stereotypes and Profiles"

See Also

[close](#) | [open](#) | [save](#)

Introduced in R2020b

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 = model.createView('All Components',...
    'Color','lightblue','Select',AnyComponent());
```

Open the Architecture Views Gallery to see the new view named 'All Components'.

```
model.openViews
```

Optionally add components to the model. Rerun the query.

```
view.runQuery()
```

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"
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"> • <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. • <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. 	"Display Component Hierarchy and Architecture Hierarchy Using Views"

See Also

systemcomposer.view.View | createView | getView | deleteView | openViews | removeQuery | modifyQuery | systemcomposer.view.ElementGroup

Topics

"Create Architecture Views Interactively"
 "Create Architectural Views Programmatically"

Introduced in R2021a

save

Package: systemcomposer.profile

Save profile as file

Syntax

```
filePath = save(profile,dirPath)
```

Description

`filePath = save(profile,dirPath)` saves a profile to disk as a file with an `.xml` extension. This function saves the file to the current directory if the optional input `dirPath` is left blank.

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

createProfile | find | editor | open | load | close | closeAll

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

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 = mSource.Architecture.addComponent('Source_Component');
mTarget = systemcomposer.createModel('Target_Model_Allocation',true);
targetComp = mTarget.Architecture.addComponent('Target_Component');

% Create the allocation set with name 'MyNewAllocation'
allocSet = systemcomposer.allocation.createAllocationSet('MyNewAllocation',...
    '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);

% Save the allocation set
allocSet.save

% 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 is a directed relationship from an element in one model to an element 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.	"Allocate Architectures in Tire Pressure Monitoring System"
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1.	"Create and Manage Allocations"
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

createAllocationSet | createScenario | deleteScenario | getScenario | load | closeAll | close | find

Topics

"Create and Manage Allocations"

Introduced in R2020b

save

Package: `systemcomposer.arch`

Save architecture model or data dictionary

Syntax

```
save(architecture)
save(dictionary)
```

Description

`save(architecture)` 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

architecture – 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"
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.	"Create an Architecture Model with Interfaces and Requirement Links"

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

loadModel | close

Topics

- “Create an Architecture Model”
- “Manage Interfaces with Data Dictionaries”

Introduced in R2019a

save

Package: `systemcomposer.analysis`

Save architecture instance

Syntax

```
save(architectureInstance, fileName)
```

Description

`save(architectureInstance, fileName)` saves an architecture instance to a MAT-file. The `save` method is part of the `systemcomposer.analysis.ArchitectureInstance` class.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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.

```
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');
```

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

architectureInstance — Architecture instance

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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an .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”

See Also

instantiate | systemcomposer.analysis.Instance | loadInstance | deleteInstance | refresh | update | lookup | iterate

Topics

“Write Analysis Function”

Introduced in R2019a

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 — Model name

character vector | string

Model name, specified as a character vector or string.

Data Types: `char` | `string`

Compatibility Considerations

saveAsModel function is not recommended

Not recommended starting in R2021b

The `saveAsModel` function is not recommended in R2021b. Use the `createArchitectureModel` function instead.

See Also

`linkToModel` | `isReference` | `createArchitectureModel` | `inlineComponent` | Reference Component

Topics

“Describe Component Behavior Using Simulink”

“Decompose and Reuse Components”

Introduced in R2019a

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 an .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, add an interface to the model's interface dictionary, and add an element. Save all interfaces defined in the model to a shared dictionary.

```
arch = systemcomposer.createModel('newModel',true);
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 — Interface dictionary

dictionary object

Interface dictionary, specified as a systemcomposer.interface.Dictionary object.

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 comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

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 the comma-separated pair consisting of 'CollisionResolutionOption' and 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"
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.	"Create an Architecture Model with Interfaces and Requirement Links"

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

[createDictionary](#) | [linkDictionary](#) | [unlinkDictionary](#) | [openDictionary](#) | [addReference](#) | [removeReference](#)

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2019b

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 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 with multiple choices.

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: 'Choice 2'

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 Condition" on page 1-531

See Also

[addChoice](#) | [getActiveChoice](#) | [getChoices](#) | [addVariantComponent](#) | [Variant Component](#)

Topics

"Create Variants"

Introduced in R2019a

setCondition

Package: systemcomposer.arch

Set condition on variant choice

Syntax

```
setCondition(variantComponent, choice, expression)
```

Description

`setCondition(variantComponent, choice, expression)` sets the variant control for a choice for the variant component.

Examples

Set Condition

Create a model, get the root architecture, create one variant component, add two choices for the variant component, set the active choice, and set a condition.

```
model = systemcomposer.createModel('archModel', true);
arch = get(model, 'Architecture');
mode = 1;
variant = addVariantComponent(arch, 'Component1');
compList = addChoice(variant, {'Choice1', 'Choice2'});
setActiveChoice(variant, compList(2));
setCondition(variant, compList(2), 'mode == 2');
```

Input Arguments

variantComponent — Variant component

variant component object

Variant component, specified as a `systemcomposer.arch.VariantComponent` object with multiple choices.

choice — Choice in variant component

component object

Choice in variant component whose control string is set by this function, specified by 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 Condition" on page 1-531

See Also

`makeVariant` | `getCondition` | `addVariantComponent` | `addChoice` | `getActiveChoice` | `setActiveChoice` | Variant Component

Topics

"Create Variants"

Introduced in R2019a

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 of parent component

`stereotype` object

Stereotype of parent component, 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`

Compatibility Considerations

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”

Introduced in R2019a

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 of parent component

`stereotype` object

Stereotype of parent component, 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`

Compatibility Considerations

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”

Introduced in R2019a

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.

```
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', 'AppliesTo', 'Connector');
connLatency.addProperty('secure', 'Type', 'boolean');
connLatency.addProperty('linkDistance', 'Type', 'double');

nodeLatency = profile.addStereotype('NodeLatency', 'Parent', ...
'LatencyProfile.LatencyBase', 'AppliesTo', 'Component');
nodeLatency.addProperty('resources', 'Type', 'double', 'DefaultValue', '1');

portLatency = profile.addStereotype('PortLatency', 'Parent', ...
'LatencyProfile.LatencyBase', 'AppliesTo', 'Port');
portLatency.addProperty('queueDepth', 'Type', 'double');
portLatency.addProperty('dummy', 'Type', 'int32');
```

Set the default component stereotype.

```
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 =
    1×1 cell array
    {'LatencyProfile.NodeLatency'}
```

Set Default Port Stereotype

Create a profile for latency characteristics.

```
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', 'AppliesTo', 'Connector');
connLatency.addProperty('secure', 'Type', 'boolean');
connLatency.addProperty('linkDistance', 'Type', 'double');

nodeLatency = profile.addStereotype('NodeLatency', 'Parent', ...
'LatencyProfile.LatencyBase', 'AppliesTo', 'Component');
nodeLatency.addProperty('resources', 'Type', 'double', 'DefaultValue', '1');

portLatency = profile.addStereotype('PortLatency', 'Parent', ...
'LatencyProfile.LatencyBase', 'AppliesTo', 'Port');
portLatency.addProperty('queueDepth', 'Type', 'double');
portLatency.addProperty('dummy', 'Type', 'int32');
```

Set the default port stereotype.

```
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 =
    1×1 cell array
```

```
{'LatencyProfile.PortLatency'}
```

Set Default Connector Stereotype

Create a profile for latency characteristics.

```
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', 'AppliesTo', 'Connector');
connLatency.addProperty('secure', 'Type', 'boolean');
connLatency.addProperty('linkDistance', 'Type', 'double');

nodeLatency = profile.addStereotype('NodeLatency', 'Parent', ...
'LatencyProfile.LatencyBase', 'AppliesTo', 'Component');
nodeLatency.addProperty('resources', 'Type', 'double', 'DefaultValue', '1');

portLatency = profile.addStereotype('PortLatency', 'Parent', ...
'LatencyProfile.LatencyBase', 'AppliesTo', 'Port');
portLatency.addProperty('queueDepth', 'Type', 'double');
portLatency.addProperty('dummy', 'Type', 'int32');
```

Set the default connector stereotype.

```
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 of parent element

stereotype object

Stereotype of parent element, specified as a `systemcomposer.profile.Stereotype` object.

elementType — Element type

'Component' | 'Port' | 'Connector' | 'Interface'

Element type, specified as 'Component', 'Port', 'Connector', or 'Interface'.

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • "Set Properties" • "Add Properties with Stereotypes" • "Set Properties for Analysis"
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • "Define Profiles and Stereotypes" • "Use Stereotypes and Profiles"

See Also

`applyStereotype` | `getDefaultElementStereotype` | `removeStereotype`

Topics

"Define Profiles and Stereotypes"

Introduced in R2021b

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 of parent component

stereotype object

Stereotype of parent component, 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`

Compatibility Considerations

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”

Introduced in R2019a

setDefaultStereotype

Package: systemcomposer.profile

Set default stereotype for profile

Syntax

```
setDefaultStereotype(profile, stereotypeName)
```

Description

setDefaultStereotype(profile, stereotypeName) sets the default stereotype for a profile. The stereotype must apply to components.

Examples

Set Default Stereotype

Create a profile for latency characteristics.

```
profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');
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.

```
profile.setDefaultStereotype('NodeLatency');
```

Create a model and apply the profile. 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.

stereotypeName – Stereotype name

character vector | string

Stereotype name, specified as a character vector or string. The stereotype must be present in the profile.

Example: 'ComponentStereotype'

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

createProfile | getDefaultStereotype | addStereotype | getStereotype | removeStereotype

Topics

“Create a Profile and Add Stereotypes”

Introduced in R2019a

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 to be renamed, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

Component | `systemcomposer.arch.ArchitecturePort` |
`systemcomposer.arch.ComponentPort`

Introduced in R2019a

setType

Package: systemcomposer.interface

Set shared type on data element

Syntax

```
setType(dataElement, type)
```

Description

setType(dataElement, type) sets a type on a data element.

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 named '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

data element object

Data element, specified as a systemcomposer.interface.DataElement object.

type — Type

data interface object | value type object

Type, specified as a systemcomposer.interface.DataInterface 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`addValueType` | `createModel` | `addInterface` | `createOwnedType` | `createInterface` | `removeInterface`

Topics

“Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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 | empty string

Interface to set, specified as a systemcomposer.interface.DataInterface, systemcomposer.ValueType, or systemcomposer.interface.PhysicalInterface 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"
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.	"Create an Architecture Model with Interfaces and Requirement Links"

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `addPhysicalInterface`

Topics

"Specify Physical Interfaces on the Ports"
 "Create Interfaces"
 "Manage Interfaces with Data Dictionaries"

Introduced in R2019a

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, 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')
```

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 — Model element

architecture object | component object | port object | connector object | physical connector object | data interface object | value type object | physical interface object

Model 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.interface.DataInterface`, `systemcomposer.ValueType`, or `systemcomposer.interface.PhysicalInterface` 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 1-561.

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

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`getProperty` | `addProperty` | `removeProperty`

Topics

"Set Properties for Analysis"

Introduced in R2019a

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.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. instance refers to the element instance on which the iteration is being performed.

Examples

Set Mass Property Value

Load the Small 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 as a character vector or string 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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an .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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

getValue | hasValue | systemcomposer.analysis.Instance

Topics

“Write Analysis Function”

Introduced in R2019a

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 named 'archModel'.

```
modelName = 'archModel';  
arch = systemcomposer.createModel(modelName, true); % Create model
```

Add a value type named 'airSpeed' to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType('airSpeed'); % Add value type
```

Set the complexity for the value type as 'complex'.

```
airSpeedType.setComplexity('complex'); % Set complexity for value type
```

Input Arguments

valueType — Value type

value type object

Value type, specified as a `systemcomposer.ValueType` object.

complexity — Complexity of value type

'real' (default) | 'complex' | 'auto'

Complexity of value type, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addElement` | `addInterface` | `addValueType` | `createInterface` | `createOwnedType`

Topics

“Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2021b

setDescription

Package: systemcomposer

Set description for value type

Syntax

```
setDescription(valueType,description)
```

Description

setDescription(valueType,description) sets the description for the designated value type.

Examples

Set Description for Value Type

Create a model named 'archModel'.

```
modelName = 'archModel';  
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add a value type named 'airSpeed' to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType('airSpeed'); % Add value type
```

Set the description for the value type as 'Maintain altitude'.

```
airSpeedType.setDescription('Maintain altitude'); % Set description for value type
```

Input Arguments

valueType — Value type

value type object

Value type, specified as a systemcomposer.ValueType object.

description — Description of value type

character vector | string

Description of value type, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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 named 'archModel'.

```
modelName = 'archModel';  
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add a value type named 'airSpeed' to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType('airSpeed'); % Add value type
```

Set the dimensions for the value type as '2'.

```
airSpeedType.setDimensions('2'); % Set dimensions for value type
```

Input Arguments

valueType – Value type

value type object

Value type, specified as a `systemcomposer.ValueType` object.

dimensions – Dimensions of value type

character vector | string

Dimensions of value type, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

Topics

“Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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 named 'archModel'.

```
modelName = 'archModel';  
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add a value type named 'airSpeed' to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType('airSpeed'); % Add value type
```

Set the maximum for the value type as '100'.

```
airSpeedType.setMaximum('100'); % Set maximum for value type
```

Input Arguments

valueType — Value type

value type object

Value type, specified as a systemcomposer.ValueType object.

maximum — Maximum of value type

character vector | string

Maximum of value type, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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 named 'archModel'.

```
modelName = 'archModel';  
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add a value type named 'airSpeed' to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType('airSpeed'); % Add value type
```

Set the minimum for the value type as '0'.

```
airSpeedType.setMinimum('0'); % Set minimum for value type
```

Input Arguments

valueType — Value type

value type object

Value type, specified as a systemcomposer.ValueType object.

minimum — Minimum of value type

character vector | string

Minimum of value type, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

Topics

“Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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 named 'archModel'.

```
modelName = 'archModel';  
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add a value type named 'airSpeed' to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType('airSpeed'); % Add value type
```

Set the units for the value type as 'm/s'.

```
airSpeedType.setUnits('m/s'); % Set units for value type
```

Input Arguments

valueType — Value type

value type object

Value type, specified as a `systemcomposer.ValueType` object.

units — Units of value type

character vector | string

Units of value type, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021b

setName

Package: systemcomposer.interface

Set name for value type, interface, or element

Syntax

```
setName(interfaceElem,name)
```

Description

setName(interfaceElem,name) sets the name for the designated value type, interface, or element.

Examples

Set Name for Data Element

Create a model named 'archModel'.

```
modelName = 'archModel';  
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add a data interface, then create an data element with the name 'x'.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add data interface  
elem = interface.addElement('x'); % Create data element
```

Set a new name for the data element as 'newName'.

```
setName(elem,'newName'); % Set new name for data element
```

Input Arguments

interfaceElem — Interface, element, or value type

data interface object | data element object | physical interface object | physical element object | value type object

Interface, element, or value type to be named, specified as a systemcomposer.interface.DataInterface, systemcomposer.interface.DataElement, systemcomposer.interface.PhysicalInterface, systemcomposer.interface.PhysicalElement, or systemcomposer.ValueType object.

name — Name

character vector | string

Name of interface, element, or value type, 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

`createModel` | `addElement` | `addInterface`

Topics

"Specify Physical Interfaces on the Ports"

"Create Interfaces"

"Manage Interfaces with Data Dictionaries"

Introduced in R2019a

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 named 'archModel'.

```
modelName = 'archModel';  
arch = systemcomposer.createModel(modelName, true); % Create model
```

Add a value type named 'airSpeed' to the interface dictionary of the model.

```
airSpeedType = arch.InterfaceDictionary.addValueType('airSpeed'); % Add value type
```

Set the data type for the value type as 'single'.

```
airSpeedType.setDataType('single'); % Set data type for value type
```

Input Arguments

valueType — Value type

value type object

Value type, specified as a `systemcomposer.ValueType` object.

type — 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.

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	<p>With an adapter, you can perform functions on the Interface Adapter dialog:</p> <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createModel` | `addValueType` | `addElement` | `addInterface` | `createInterface` | `createOwnedType`

Topics

“Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1 .	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

`createScenario` | `deleteScenario` | `getScenario` | `load` | `systemcomposer.allocation.AllocationSet.find` | `closeAll` | `close`

Topics

“Create and Manage Allocations”

Introduced in R2020b

unlinkDictionary

Package: systemcomposer.arch

Unlink data dictionary from architecture model

Syntax

```
unlinkDictionary(modelObject)
```

Description

unlinkDictionary(modelObject) 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

modelObject — Architecture model

model object

Architecture model from which the dictionary link is to be removed, 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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	"Compose Architecture Visually"
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	"Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"
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.	"Create an Architecture Model with Interfaces and Requirement Links"

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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

[linkDictionary](#) | [saveToDictionary](#) | [createDictionary](#) | [addReference](#) | [removeReference](#)

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2019a

update

Package: systemcomposer.analysis

Update architecture model

Syntax

```
update(architectureInstance)
```

Description

`update(architectureInstance)` updates a specification model to mirror the changes in the architecture instance `architectureInstance`. The `update` method is part of the `systemcomposer.analysis.ArchitectureInstance` class.

This function is part of the instance API that you can use to analyze the model iteratively, element by element. `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.

```
profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency', 'Type', 'double');
latencybase.addProperty('dataRate', 'Type', 'double', 'DefaultValue', '10');
```

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

architectureInstance — Architecture instance

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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an <code>.MAT</code> 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”

See Also

instantiate | systemcomposer.analysis.Instance | loadInstance | deleteInstance | save | lookup | iterate | refresh

Topics

“Write Analysis Function”

Introduced in R2019a

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 Simulink Requirements 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 to make full use of the Simulink® Requirements™ functionality.

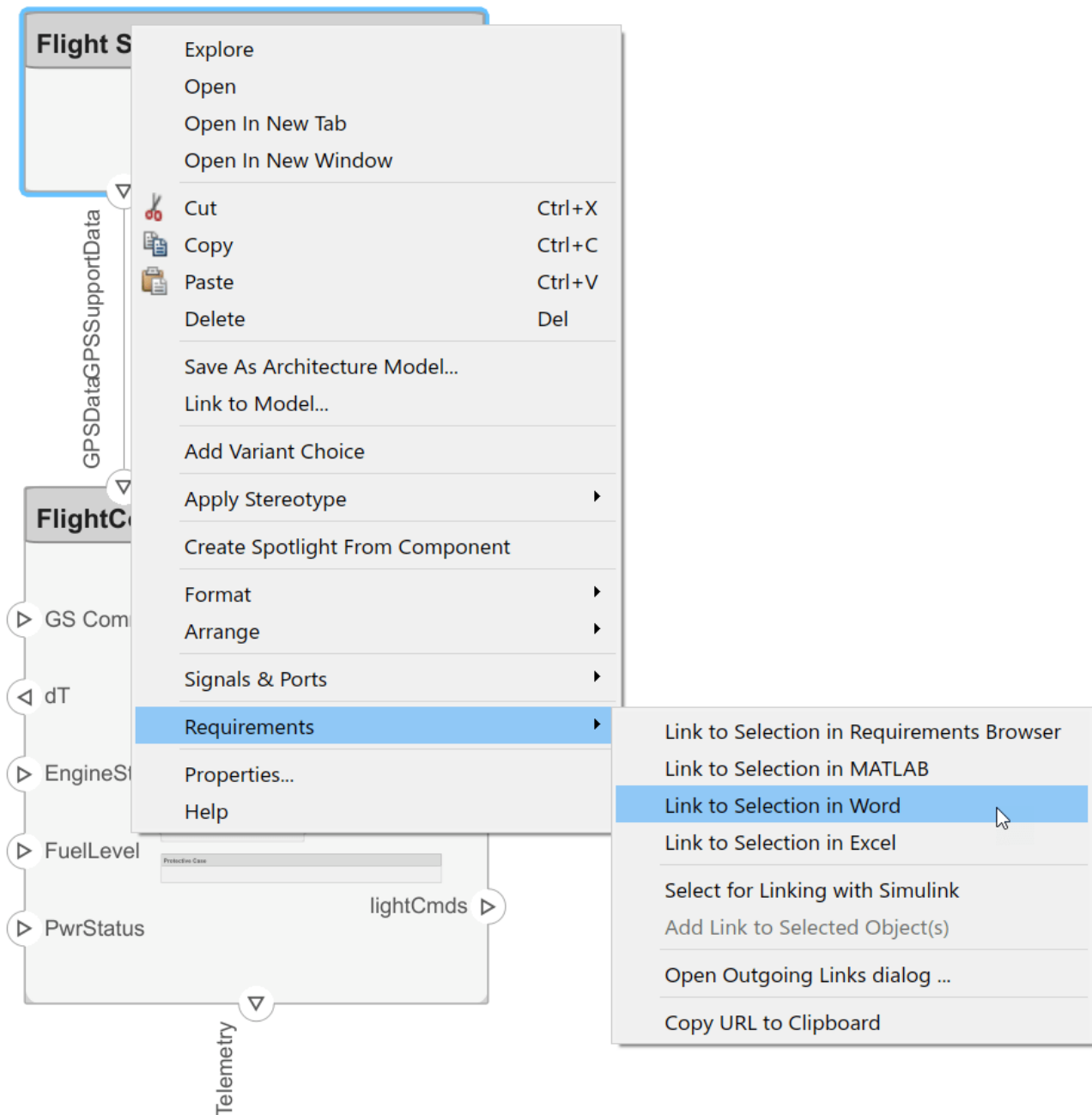
```
model = systemcomposer.openModel('reqImportExample');
```

Note: Importing or linking requirements may not work with a web-based Microsoft® Office file stored in SharePoint or OneDrive. Use a local copy of the file.

Import Requirement Links from Word File

Open the Microsoft® Word file `Functional_Requirements.docx` with the requirements listed. Highlight the requirement to link.

In the model, select the component to which to link the requirement. Right-click the component and select **Requirements > Link to Selection in Word**.



Export Model and Save to External File

Export the model and save to an external file.

```
exportedSet = systemcomposer.exportModel('reqImportExample');
SaveToExcel('exportedModel',exportedSet);
```

Import Requirement Links from File and Import to Model

Use the external file to import requirement links into another model.

```
structModel = ImportModelFromExcel('exportedModel.xls', 'Components', 'Ports', ...  
'Connections', 'PortInterfaces', 'RequirementLinks');  
structModel.readTableFromExcel
```

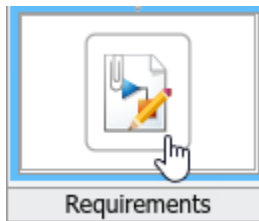
```
systemcomposer.importModel('reqNewExample', structModel.Components, ...  
structModel.Ports, structModel.Connections, structModel.Interfaces, structModel.RequirementLinks);
```

Update Links to Reference Requirements

To integrate the requirement links to the model, update references within the model.

```
systemcomposer.updateLinksToReferenceRequirements("reqNewExample", "linktype_rmi_word", "Functional
```

Open the **Requirements** perspective from the bottom right corner of the model palette to view the requirements.



Input Arguments

modelName — Name of model

character vector | string

Name of model, specified as a character vector or string.

Data Types: char | string

linkDomain — Link domain

character vector | string

Link domain, specified as a character vector or string. See “Custom Link Types” (Simulink Requirements) 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 Manager on an architecture model or through custom views. Assign test cases to requirements using the Test Manager for verification and validation.	“Link and Trace Requirements”
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 <code>.slreqx</code> files. You can create a new requirement set and author requirements using Simulink Requirements, or import requirements from supported third-party tools.	“Manage Requirements”
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 <code>.slmx</code> files.	<ul style="list-style-type: none"> • “Create an Architecture Model with Interfaces and Requirement Links” • “Update Reference Requirement Links from Imported File” on page 1-606

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. The Interface Editor is accessible in System Composer test harness models to enable behavior testing and implementation-independent interface testing.	<ul style="list-style-type: none">• “Verify and Validate Requirements Using Test Harnesses on Components”• “Create a Test Harness” (Simulink Test)

See Also

`importModel` | `exportModel`

Topics

“Link and Trace Requirements”

“Manage Requirements”

“Import and Export Architecture Models”

“Custom Link Types” (Simulink Requirements)

Introduced in R2020b

Classes

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 allocations.

```
% Create two allocations between four elements in
% the default scenario, 'Scenario 1'.
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 allocation scenario or allocation

Examples

Allocate Architectures in Tire Pressure Monitoring System

Use allocations to analyze a tire pressure monitoring system.

Overview

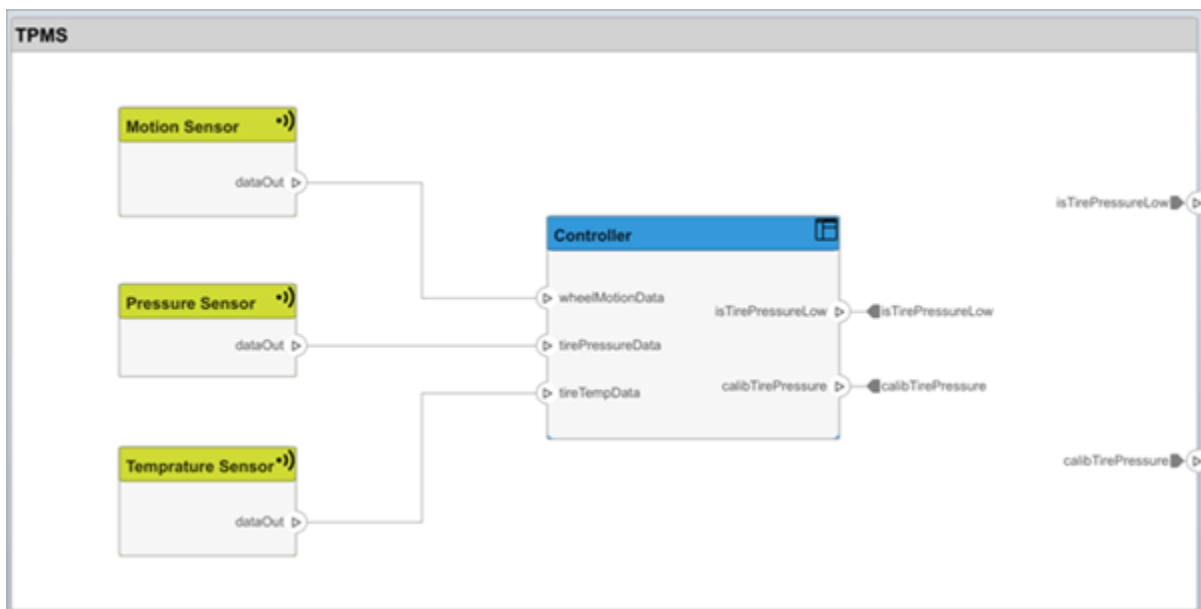
In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system 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.
- 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.
- 3 Platform Architecture — Describes the physical hardware needed for the system at a high level.

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`. 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-->	callbTirePressure-->1	callbTirePressure-->1	isTirePressureLow-->	isTirePressureLow-->	callbTirePressure-->1	isTirePressureLow-->	callbTirePressure-->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, 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
Calculate if pressure is low	1	0	0	0
Report Tire Pressure Levels	1	0	0	0
Calculate Tire Pressure	0	1	0	0
Report Low Tire Pressure	1	0	0	0
Measure temprature of tire	0	0	0	1
Measure rotations	0	1	0	0
Measure pressure on tire	0	0	1	0
Measure Tire Pressure	0	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
end
end

```

More About

Definitions

Term	Definition	Application	More Information
allocation	An allocation is a directed relationship from an element in one model to an element 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.	"Allocate Architectures in Tire Pressure Monitoring System"
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1 .	"Create and Manage Allocations"
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

getAllocatedFrom | getAllocation | getAllocatedTo | allocate | getScenario

Topics

"Create and Manage Allocations"

Introduced in R2020b

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 scenario.

```
scenario = createScenario(myAllocationSet)
```

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 allocation scenario or allocation

Examples

Allocate Architectures in Tire Pressure Monitoring System

Use allocations to analyze a tire pressure monitoring system.

Overview

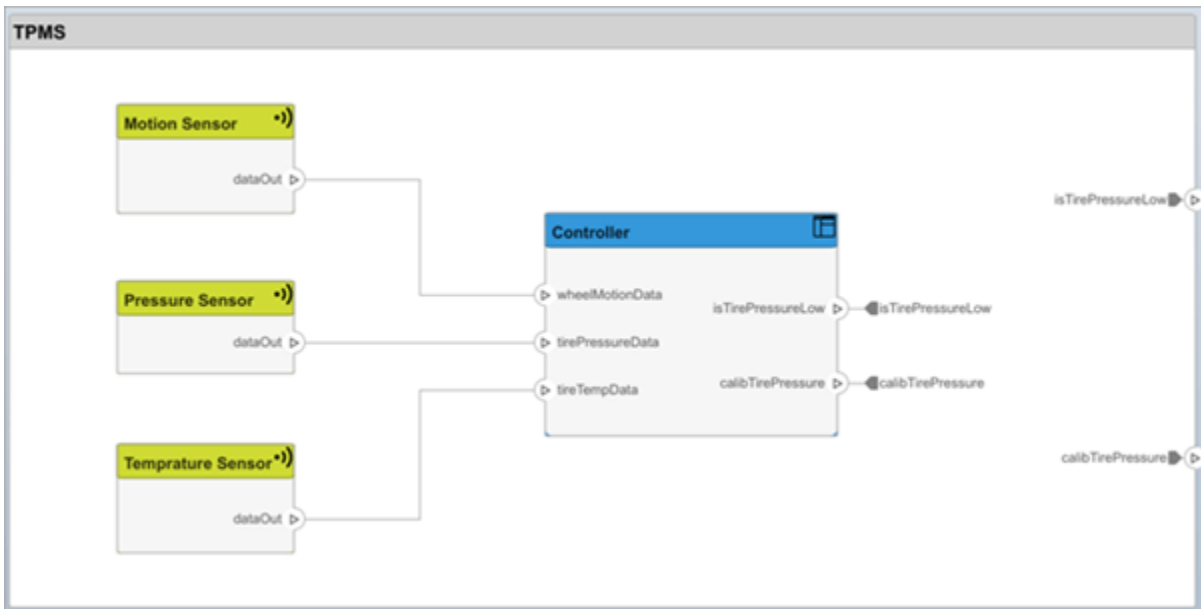
In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system 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.
- 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.
- 3 Platform Architecture — Describes the physical hardware needed for the system at a high level.

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. 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
Calculate if pressure is low	1	0	0	0
Report Tire Pressure Levels	1	0	0	0
Calculate Tire Pressure	0	1	0	0
Report Low Tire Pressure	1	0	0	0
Measure temprature of tire	0	0	0	1
Measure rotations	0	1	0	0
Measure pressure on tire	0	0	1	0
Measure Tire Pressure	0	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 is a directed relationship from an element in one model to an element 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.	"Allocate Architectures in Tire Pressure Monitoring System"
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1 .	"Create and Manage Allocations"
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	"Create and Manage Allocations"

See Also

createScenario

Topics

“Create and Manage Allocations”

Introduced in R2020b

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 and view it.

```
% Create the allocation set with name MyNewAllocation.
systemcomposer.allocation.createAllocationSet('MyNewAllocation', ...
    'Source_Model_Allocation', 'Target_Model_Allocation');

% Open the allocation editor
systemcomposer.allocation.editor()
```

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

createScenario	Create new empty allocation scenario
getScenario	Get allocation scenario
deleteScenario	Delete allocation scenario
synchronizeChanges	Synchronize changes of models in allocation set
find	Find loaded allocation set
save	Save allocation set as file
close	Close allocation set
closeAll	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:

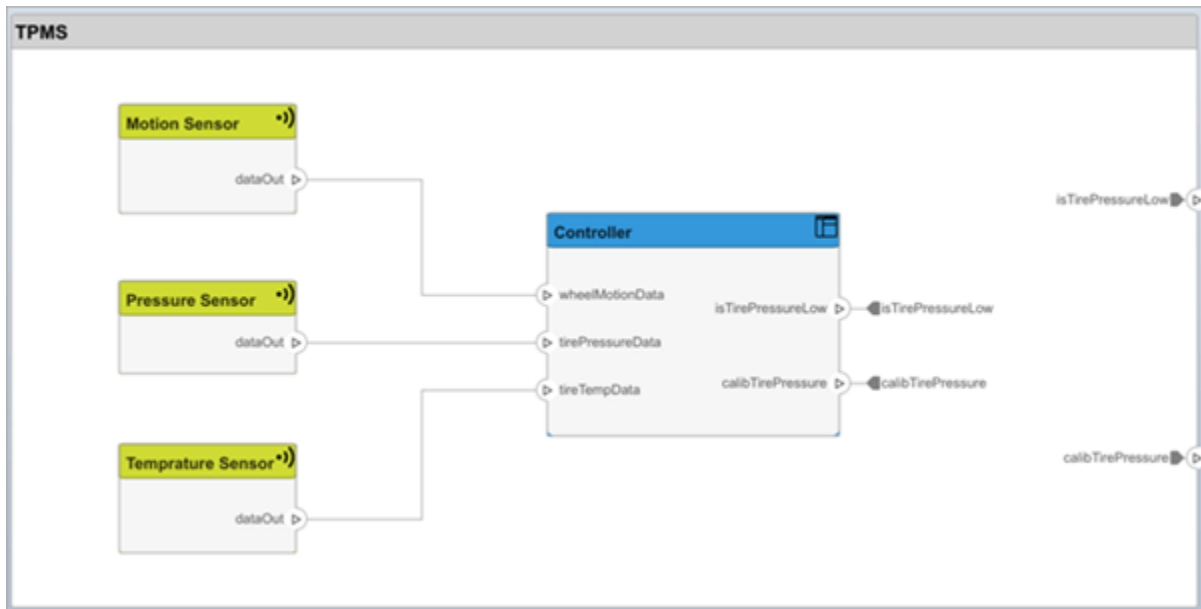
- 1 Functional Architecture — Describes the system in terms of its high-level functions. The connections show dependencies between functions.
- 2 Logical Architecture — Describes the system in terms of its logical components and how data is exchanged between them. Additionally, this architecture specifies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

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

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`. 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), 'RowNames', functionNames);
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=8x4 table
```

	Supplier A	Supplier B	Supplier C	Supplier D
Calculate if pressure is low	1	0	0	0
Report Tire Pressure Levels	1	0	0	0
Calculate Tire Pressure	0	1	0	0
Report Low Tire Pressure	1	0	0	0
Measure temprature of tire	0	0	0	1
Measure rotations	0	1	0	0
Measure pressure on tire	0	0	1	0
Measure Tire Pressure	0	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 is a directed relationship from an element in one model to an element 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.	“Allocate Architectures in Tire Pressure Monitoring System”
allocation scenario	An allocation scenario contains a set of allocations between a source and target model.	Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called Scenario 1 .	“Create and Manage Allocations”
allocation set	An allocation set consists of one more allocation scenarios which describe various allocations between a source and target model.	Create an allocation set with allocation scenarios.	“Create and Manage Allocations”

See Also

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

Topics

“Create and Manage Allocations”

Introduced in R2020b

systemcomposer.analysis.ArchitectureInstance

Architecture in analysis instance

Description

An ArchitectureInstance object represents an instance of an architecture.

Creation

Create an instance of an architecture.

```
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 architecture 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 base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency', 'Type', 'double');
latencybase.addProperty('dataRate', 'Type', 'double', 'DefaultValue', '10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency', 'Parent', ...
'LatencyProfileC.LatencyBase');
connLatency.addProperty('secure', 'Type', 'boolean', 'DefaultValue', 'true');
connLatency.addProperty('linkDistance', 'Type', 'double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency', 'Parent', ...
'LatencyProfileC.LatencyBase');
nodeLatency.addProperty('resources', 'Type', 'double', 'DefaultValue', '1');

% Add 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); % Create new model
arch = model.Architecture;
```

```

model.applyProfile('LatencyProfileC'); % Apply profile to model

% Create components, ports, and connections
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
planningPorts = addPort(components(2).Architecture,{'SensorData','MotionCommand'},{'in','out'});
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
c_sensorData = connect(arch,components(1),components(2));
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));

% Clean up 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
```

Clean Up

Uncomment the following code and run to clean up the artifacts created by this example.

```
% bdclose('archModel')
% systemcomposer.profile.Profile.closeAll
```

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

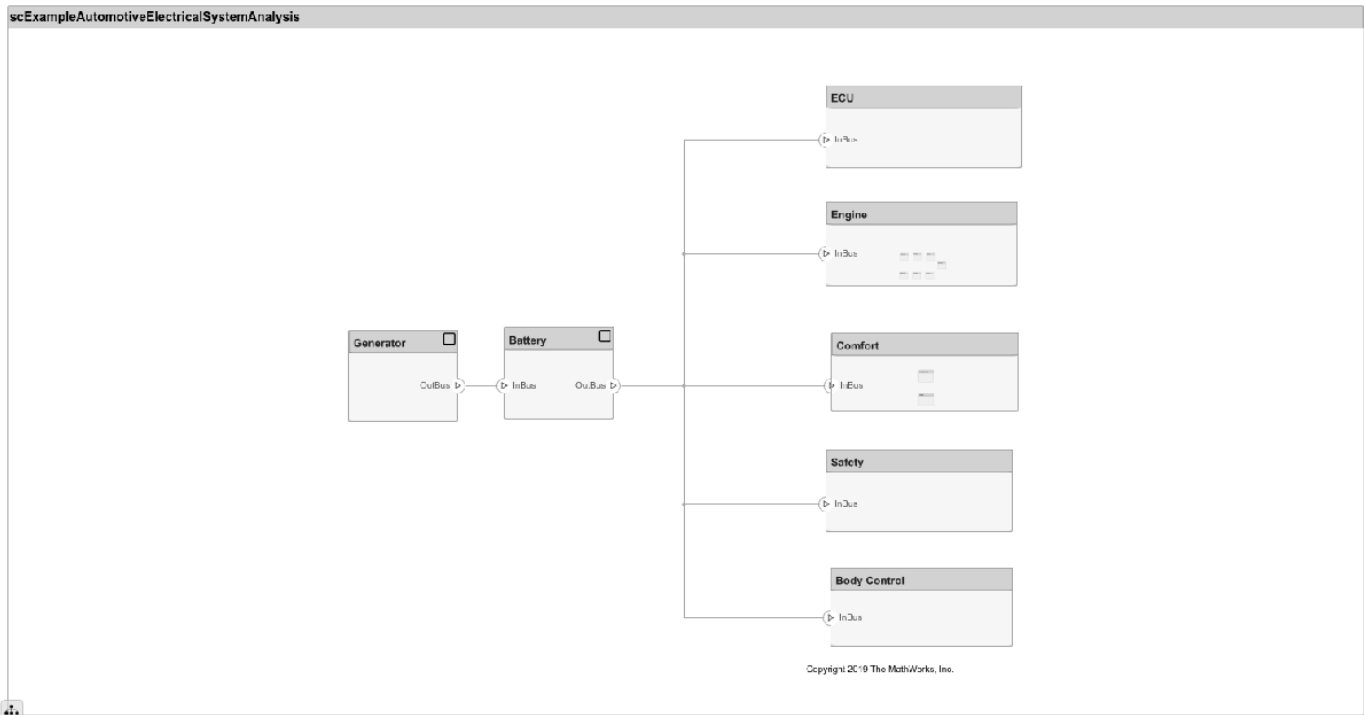
The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set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

```
archModel = systemcomposer.openModel('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 specifed battery is sufficient to start the car at 0 F.
```

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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”

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 an .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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. <p>System Composer models are stored as SLX files.</p>	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

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

Topics

"Write Analysis Function"

Introduced in R2019a

systemcomposer.analysis.ComponentInstance

Component in analysis instance

Description

A ComponentInstance object represents an instance of a component.

Creation

Create an instance of an architecture.

```
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 base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency', 'Type', 'double');
latencybase.addProperty('dataRate', 'Type', 'double', 'DefaultValue', '10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency', 'Parent', ...
'LatencyProfileC.LatencyBase');
connLatency.addProperty('secure', 'Type', 'boolean', 'DefaultValue', 'true');
connLatency.addProperty('linkDistance', 'Type', 'double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency', 'Parent', ...
'LatencyProfileC.LatencyBase');
nodeLatency.addProperty('resources', 'Type', 'double', 'DefaultValue', '1');

% Add 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); % Create new model
arch = model.Architecture;
```

```

model.applyProfile('LatencyProfileC'); % Apply profile to model

% Create components, ports, and connections
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
planningPorts = addPort(components(2).Architecture,{'SensorData','MotionCommand'},{'in','out'});
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
c_sensorData = connect(arch,components(1),components(2));
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));

% Clean up 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

```

Clean Up

Uncomment the following code and run to clean up the artifacts created by this example.

```
% bdclose('archModel')
% systemcomposer.profile.Profile.closeAll
```

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

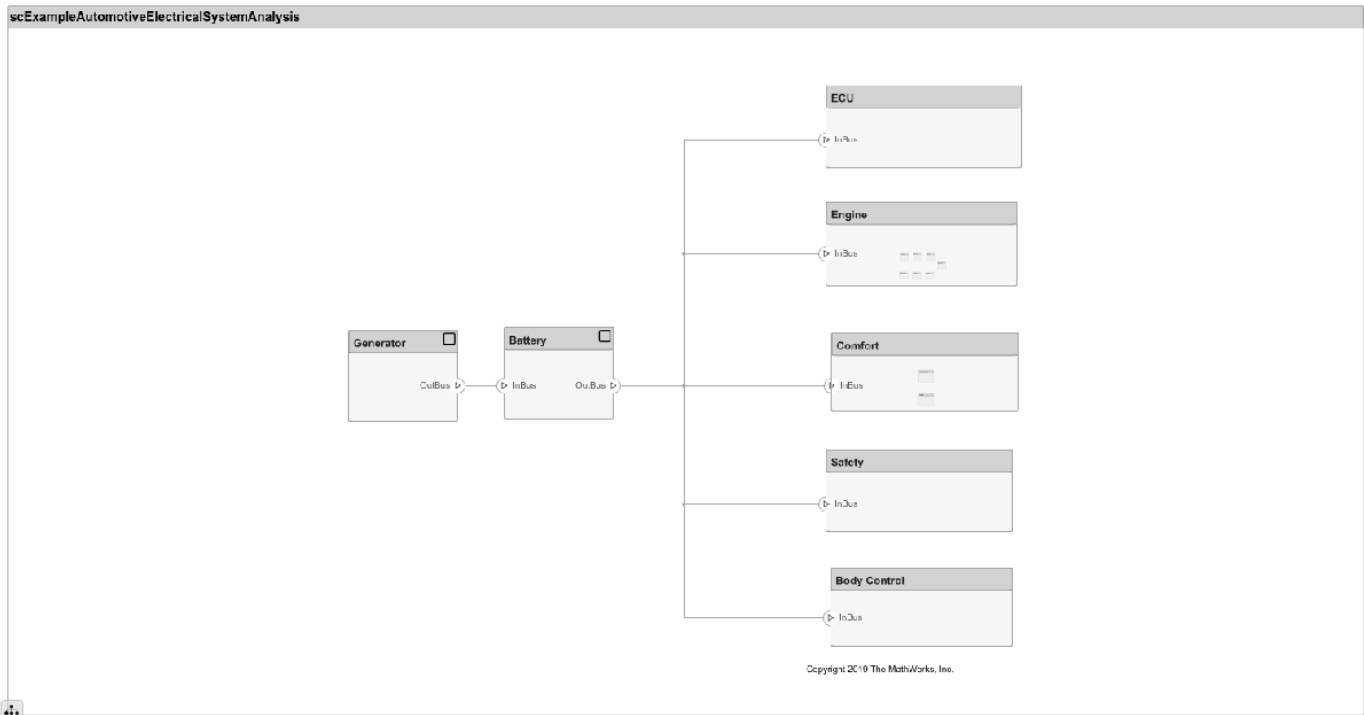
The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set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

```
archModel = systemcomposer.openModel('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 specied battery is sufficient to start the car at 0 F.
```



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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”

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 an .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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

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"

Introduced in R2019a

systemcomposer.analysis.ConnectorInstance

Connector in analysis instance

Description

A ConnectorInstance object represents an instance of a connector.

Creation

Create an instance of an architecture.

```
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.

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

getValue	Get value of property from element instance
setValue	Set value of property for element instance
hasValue	Find if element instance has property value
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 base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency', 'Type', 'double');
latencybase.addProperty('dataRate', 'Type', 'double', 'DefaultValue', '10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency', 'Parent', ...
'LatencyProfileC.LatencyBase');
connLatency.addProperty('secure', 'Type', 'boolean', 'DefaultValue', 'true');
connLatency.addProperty('linkDistance', 'Type', 'double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency', 'Parent', ...
'LatencyProfileC.LatencyBase');
nodeLatency.addProperty('resources', 'Type', 'double', 'DefaultValue', '1');

% Add 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); % Create new model
arch = model.Architecture;

model.applyProfile('LatencyProfileC'); % Apply profile to model

% Create components, ports, and connections
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
planningPorts = addPort(components(2).Architecture,{'SensorData','MotionCommand'},{'in','out'});
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
c_sensorData = connect(arch,components(1),components(2));
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));

% Clean up 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.queueD
```

```
defaultQueueDepth = 4.2900
```

Clean Up

Uncomment the following code and run to clean up the artifacts created by this example.

```
% bdclose('archModel')  
% systemcomposer.profile.Profile.closeAll
```

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:

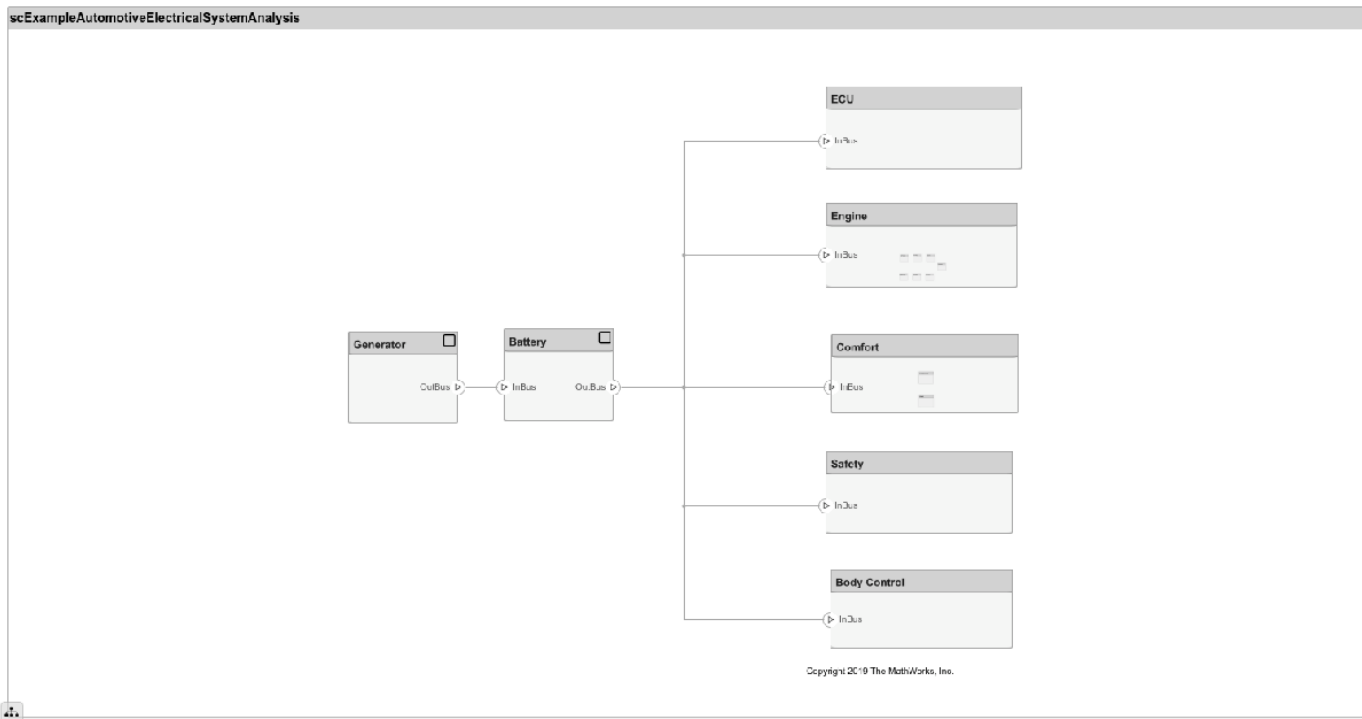
- Total KeyOffLoad.
- Number of days required for KeyOffLoad to discharge 30% of the battery.
- Total CrankingInRush current.
- Total Cranking current.
- Ability of the battery to start the vehicle at 0°F based on the battery cold cranking amps (CCA). The discharge time is computed based on Puekert coefficient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

Load Model and Run Analysis

```
archModel = systemcomposer.openModel('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.



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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”

Term	Definition	Application	More Information
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”
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 an .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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”

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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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”

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”

Introduced in R2019a

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.

```
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 base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency','Type','double');
latencybase.addProperty('dataRate','Type','double','DefaultValue','10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency','Parent',...
'LatencyProfileC.LatencyBase');
connLatency.addProperty('secure','Type','boolean','DefaultValue','true');
connLatency.addProperty('linkDistance','Type','double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency','Parent',...
'LatencyProfileC.LatencyBase');
nodeLatency.addProperty('resources','Type','double','DefaultValue','1');

% Add 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); % Create new model
arch = model.Architecture;

model.applyProfile('LatencyProfileC'); % Apply profile to model

% Create components, ports, and connections
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
planningPorts = addPort(components(2).Architecture,{'SensorData','MotionCommand'},{'in','out'});
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
c_sensorData = connect(arch,components(1),components(2));
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));

% Clean up 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
```

Clean Up

Uncomment the following code and run to clean up the artifacts created by this example.

```

% bdclose('archModel')
% systemcomposer.profile.Profile.closeAll

```

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

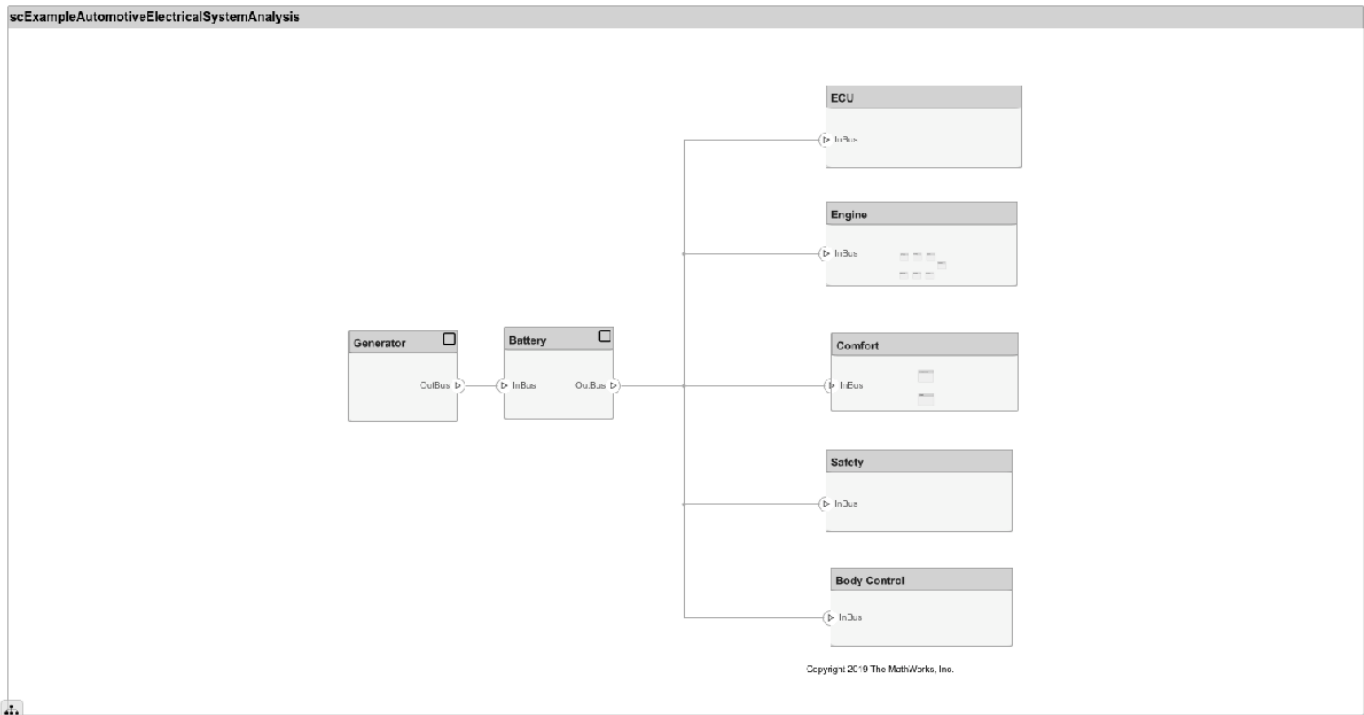
The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set 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 Puckert coefficient (k), which describes the relationship between the rate of discharge and the available capacity of the battery.

Load Model and Run Analysis

```
archModel = systemcomposer.openModel('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.
```



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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”

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 an .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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. <p>System Composer models are stored as SLX files.</p>	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

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"

Introduced in R2019a

systemcomposer.analysis.PortInstance

Port in analysis instance

Description

A PortInstance object represents an instance of a port.

Creation

Create an instance of an architecture.

```
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 base stereotype with properties
latencybase = profile.addStereotype('LatencyBase');
latencybase.addProperty('latency', 'Type', 'double');
latencybase.addProperty('dataRate', 'Type', 'double', 'DefaultValue', '10');

% Add connector stereotype with properties
connLatency = profile.addStereotype('ConnectorLatency', 'Parent', ...
'LatencyProfileC.LatencyBase');
connLatency.addProperty('secure', 'Type', 'boolean', 'DefaultValue', 'true');
connLatency.addProperty('linkDistance', 'Type', 'double');

% Add component stereotype with properties
nodeLatency = profile.addStereotype('NodeLatency', 'Parent', ...
'LatencyProfileC.LatencyBase');
nodeLatency.addProperty('resources', 'Type', 'double', 'DefaultValue', '1');

% Add 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); % Create new model
arch = model.Architecture;

model.applyProfile('LatencyProfileC'); % Apply profile to model

% Create components, ports, and connections
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
planningPorts = addPort(components(2).Architecture,{'SensorData','MotionCommand'},{'in','out'});
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
c_sensorData = connect(arch,components(1),components(2));
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));

% Clean up 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

```

Clean Up

Uncomment the following code and run to clean up the artifacts created by this example.

```
% bdclose('archModel')
% systemcomposer.profile.Profile.closeAll
```

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

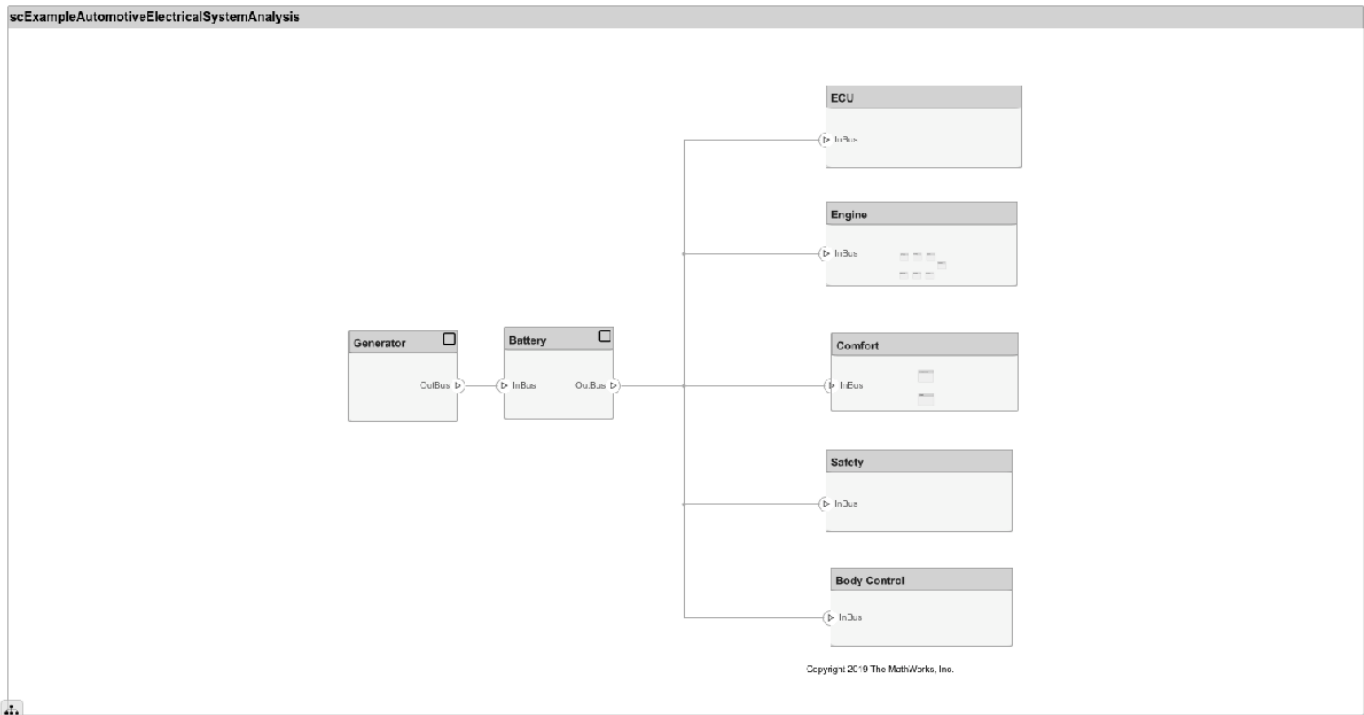
The generator charges the battery while the engine is running. The battery and the generator support the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties set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

```
archModel = systemcomposer.openModel('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 specifed battery is sufficient to start the car at 0 F.
```



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"> “Analyze an Architecture Model with an Analysis Function” “Analyze Architecture”
analysis function	An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using properties of each element in the model instance.	Use an analysis function to calculate the result of an analysis.	“Write Analysis Function”

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 an .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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

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"

Introduced in R2019a

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 and get the root architecture.

```
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.

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 of architecture, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: double

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of architecture, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: double

Object Functions

<code>addComponent</code>	Add components to architecture
<code>addVariantComponent</code>	Add variant components to architecture
<code>addPort</code>	Add ports to architecture
<code>connect</code>	Create architecture model connections
<code>applyStereotype</code>	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 component
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

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, and value types, then add components, ports, and connections. 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. 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 S');
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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','interfaces');
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.

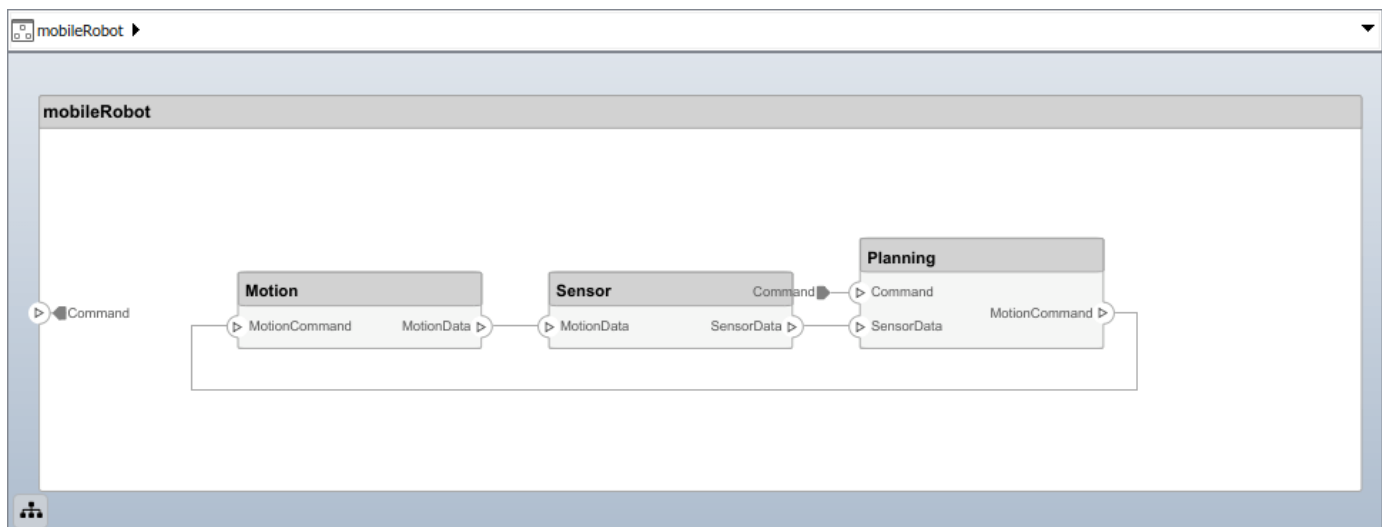
```
model.save
```

Open the model.

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

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. 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');
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 control');
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 interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn');

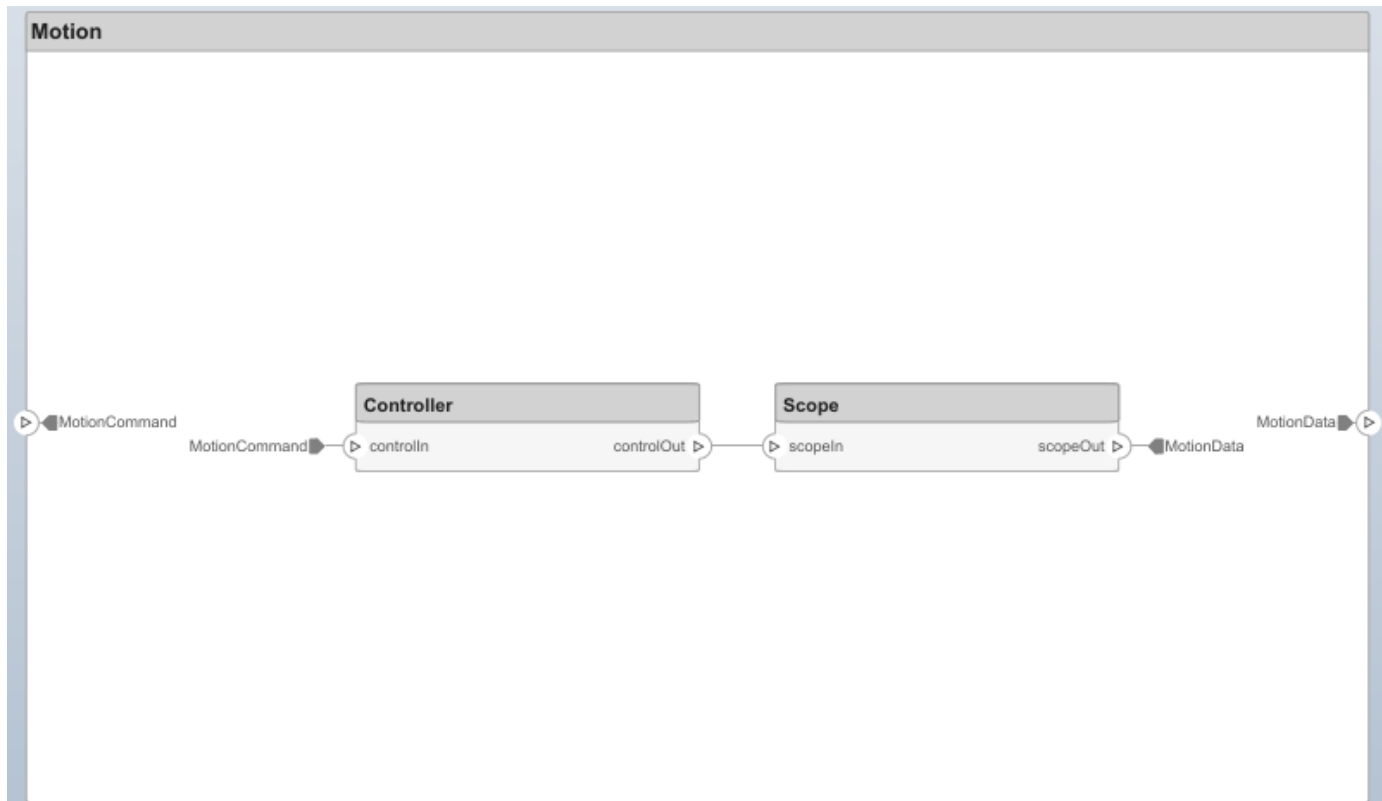
```

Save the model.

```
model.save
```

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

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



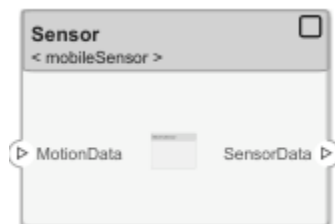
Create Model Reference

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 **Sensor** component into a reference component to reference the new model. To add additional ports on the **Sensor** component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

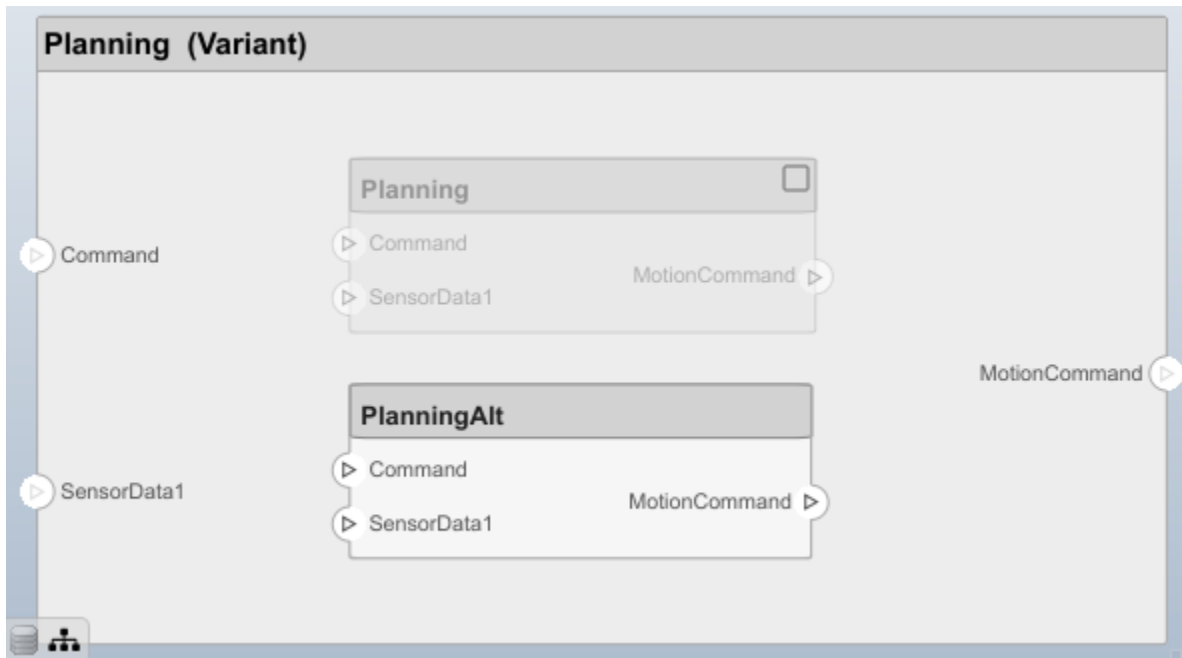
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

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

Topics

"Create an Architecture Model"

Introduced in R2019a

systemcomposer.arch.ArchitecturePort

Input and output ports of architecture

Description

An `ArchitecturePort` object represents the input and output 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.

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

The `addPort` method is the constructor for the `systemcomposer.arch.ArchitecturePort` class.

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'`

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

Interface associated with port, specified as a `systemcomposer.interface.DataInterface` or `systemcomposer.ValueType` 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 of architecture port, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: `double`

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of architecture port, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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
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 component
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, and value types, then add components, ports, and connections. 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. 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 S');
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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','interfaces');
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.

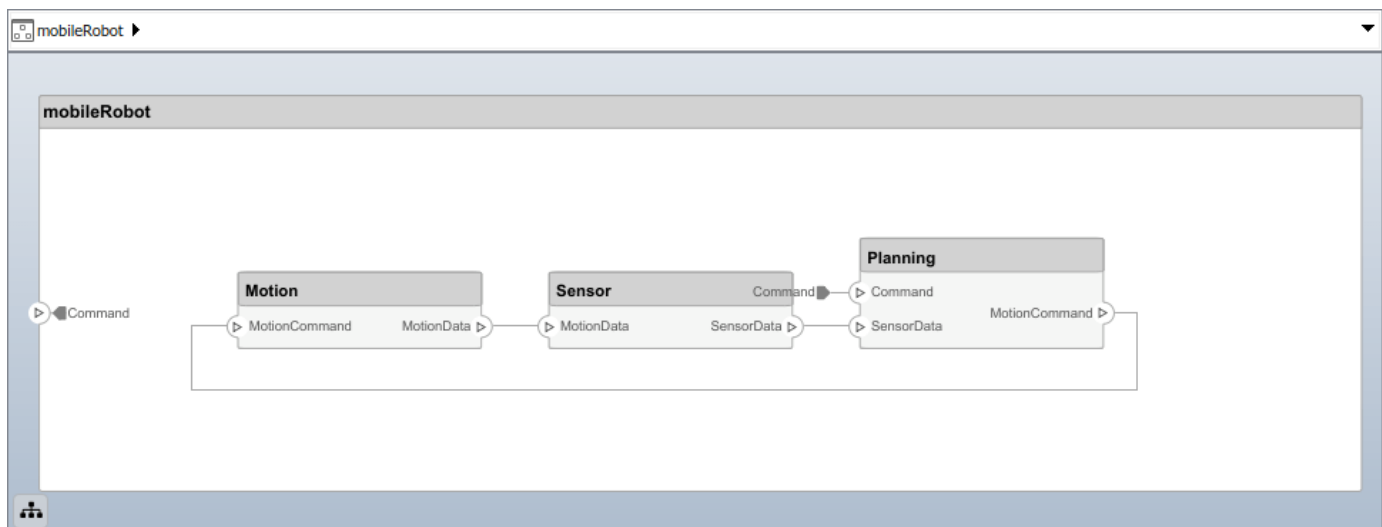
```
model.save
```

Open the model.

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

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. 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');
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 control');
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 interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn')

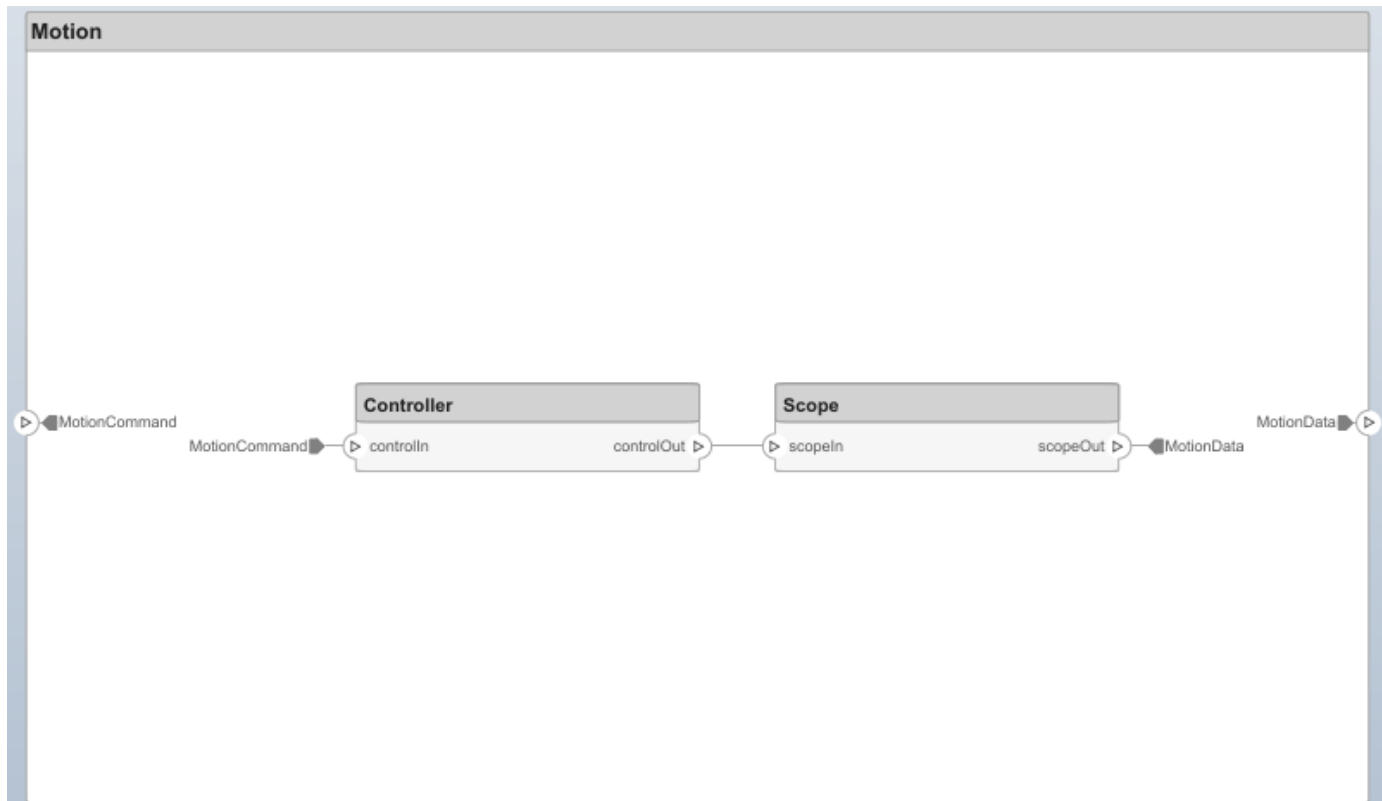
```

Save the model.

```
model.save
```

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

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



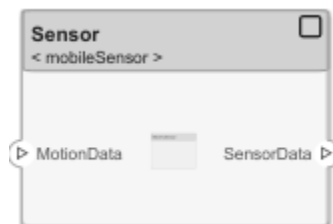
Create Model Reference

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 `Sensor` component into a reference component to reference the new model. To add additional ports on the `Sensor` component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.sldd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

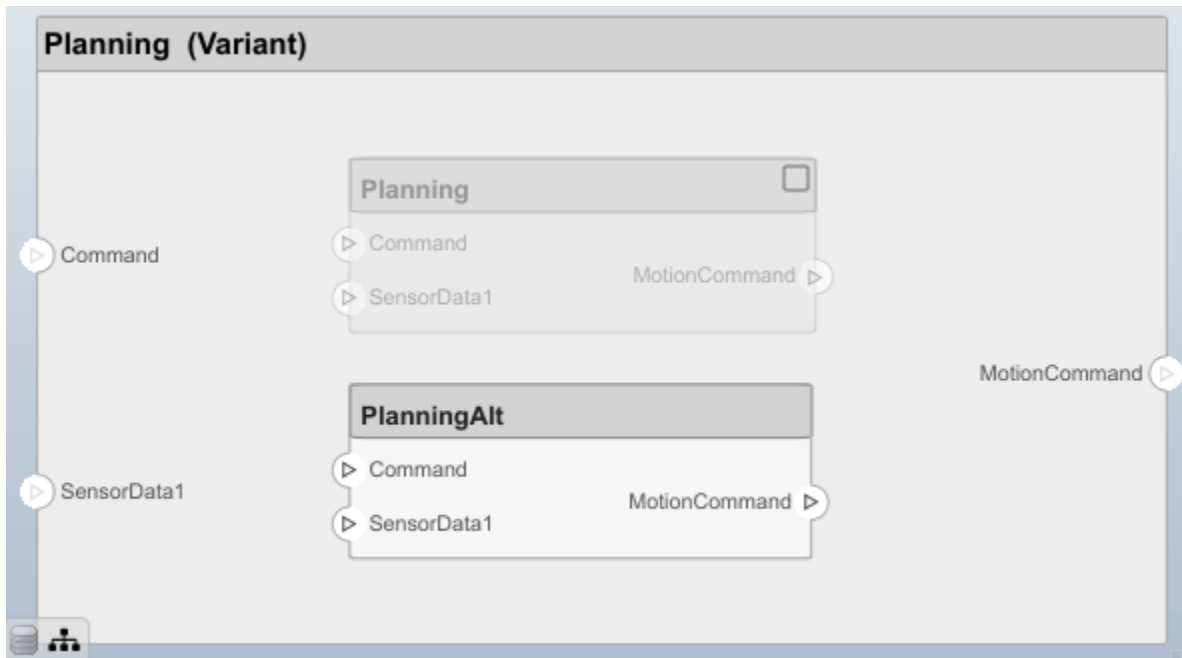
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

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

Topics

"Create an Architecture Model"

Introduced in R2019a

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.

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

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 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 of component, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: double

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of component, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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 component
<code>getStereotypeProperties</code>	Get stereotype property names on element

applyStereotype	Apply stereotype to architecture model element
getStereotypes	Get stereotypes applied on element of architecture model
removeStereotype	Remove stereotype from model element
isProtected	Find if component referenced model is protected
isReference	Find if component is referenced to another model
connect	Create architecture model connections
getPort	Get port from component
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, and value types, then add components, ports, and connections. 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. 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 S');
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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','interfaces');
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.

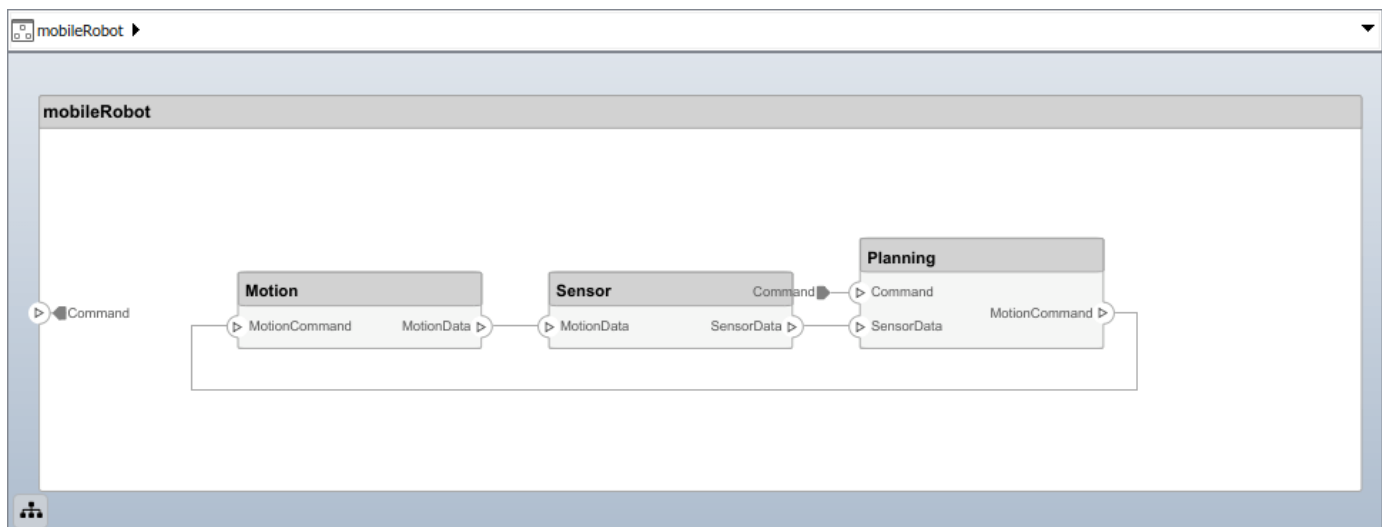
```
model.save
```

Open the model.

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

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. 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');
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 control');
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 interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn');

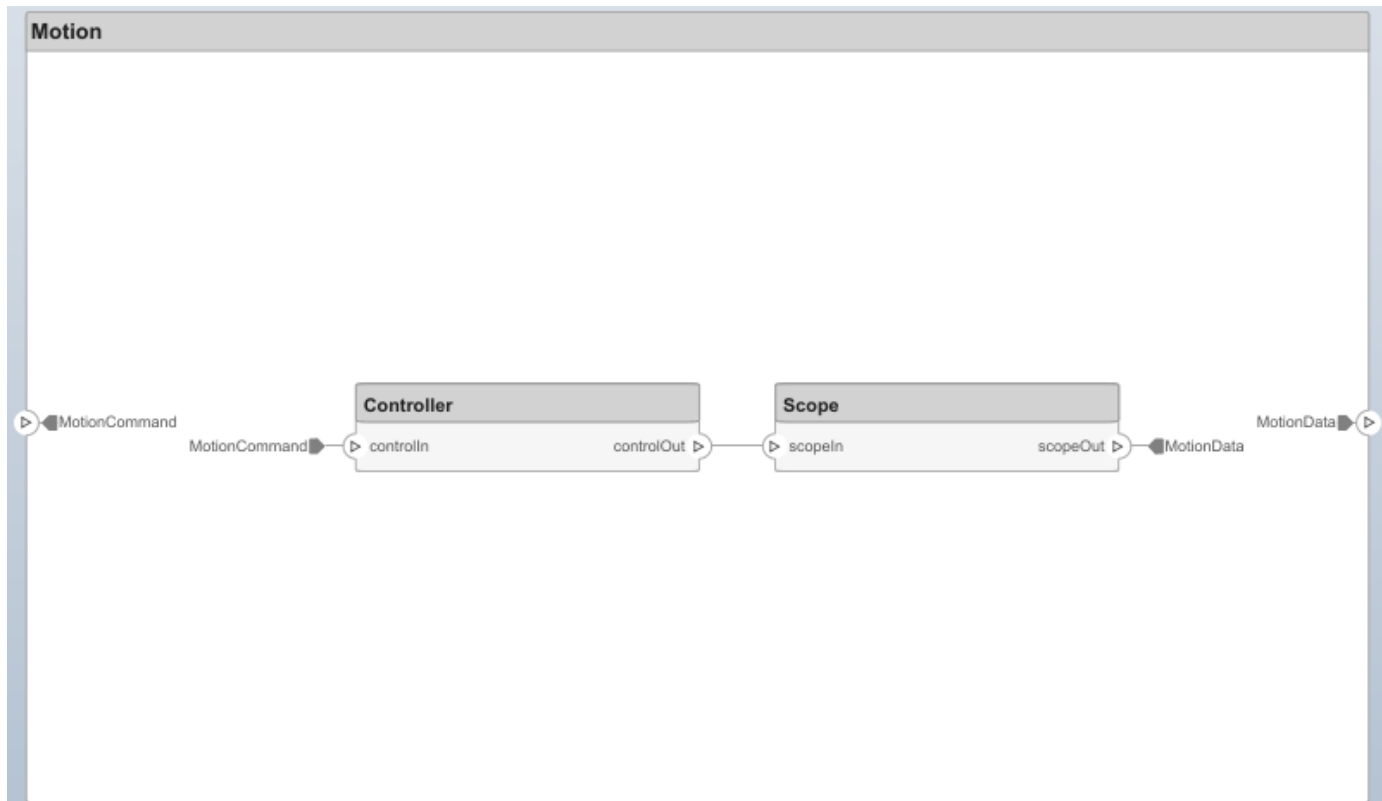
```

Save the model.

```
model.save
```

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

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



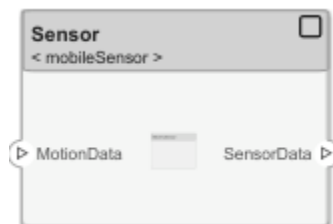
Create Model Reference

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 **Sensor** component into a reference component to reference the new model. To add additional ports on the **Sensor** component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

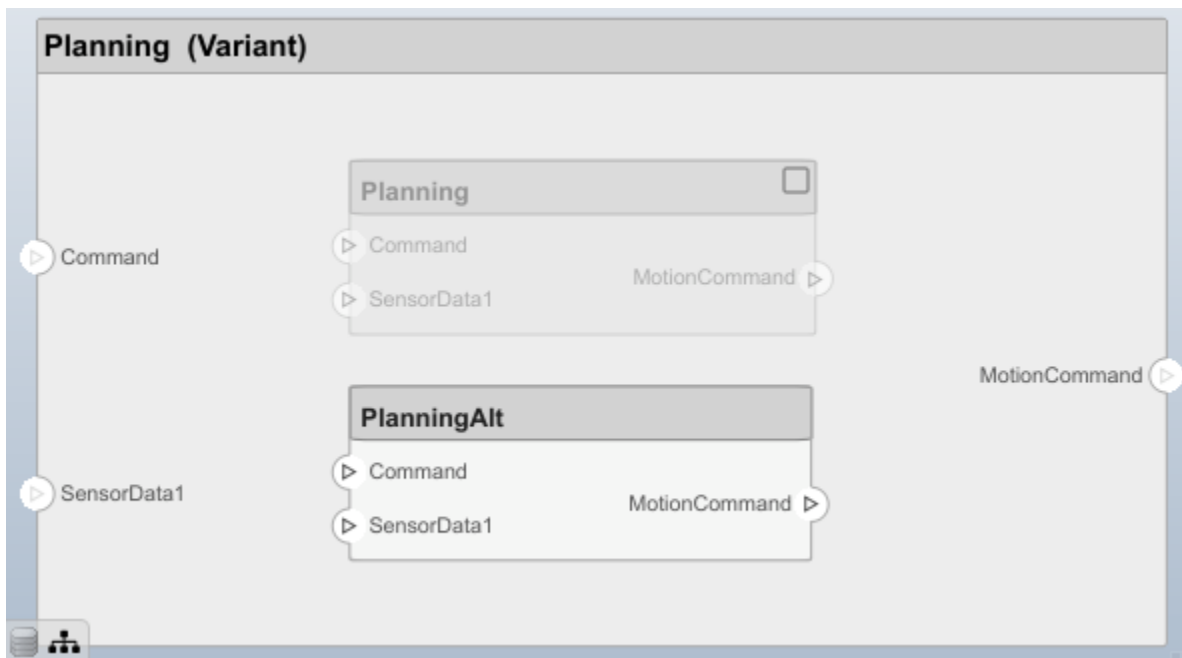
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

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

Topics

"Create an Architecture Model"

Introduced in R2019b

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 of connector, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: `double`

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of connector, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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 component
<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, and value types, then add components, ports, and connections. 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;
```

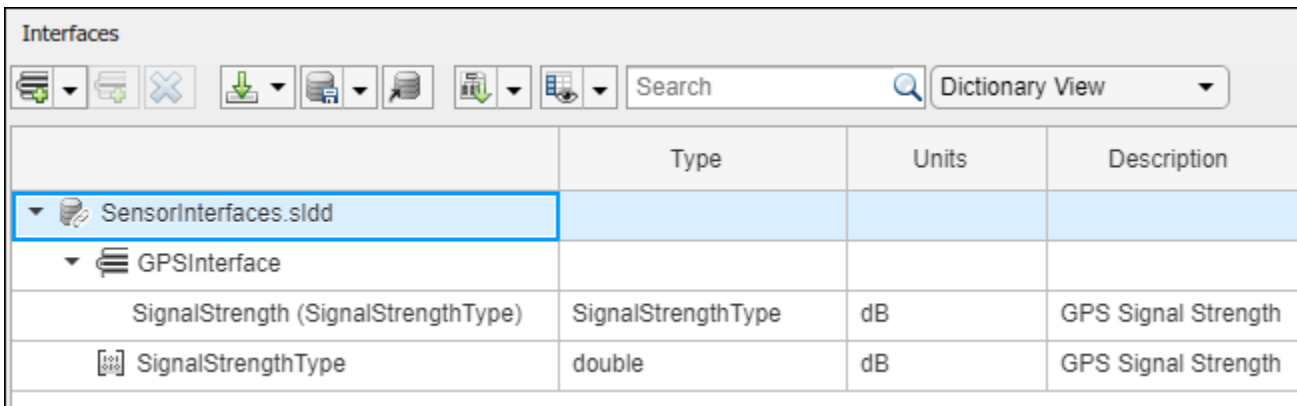
Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Link the data dictionary to the model.

```
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);
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)


componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)

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','interfaces');
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.

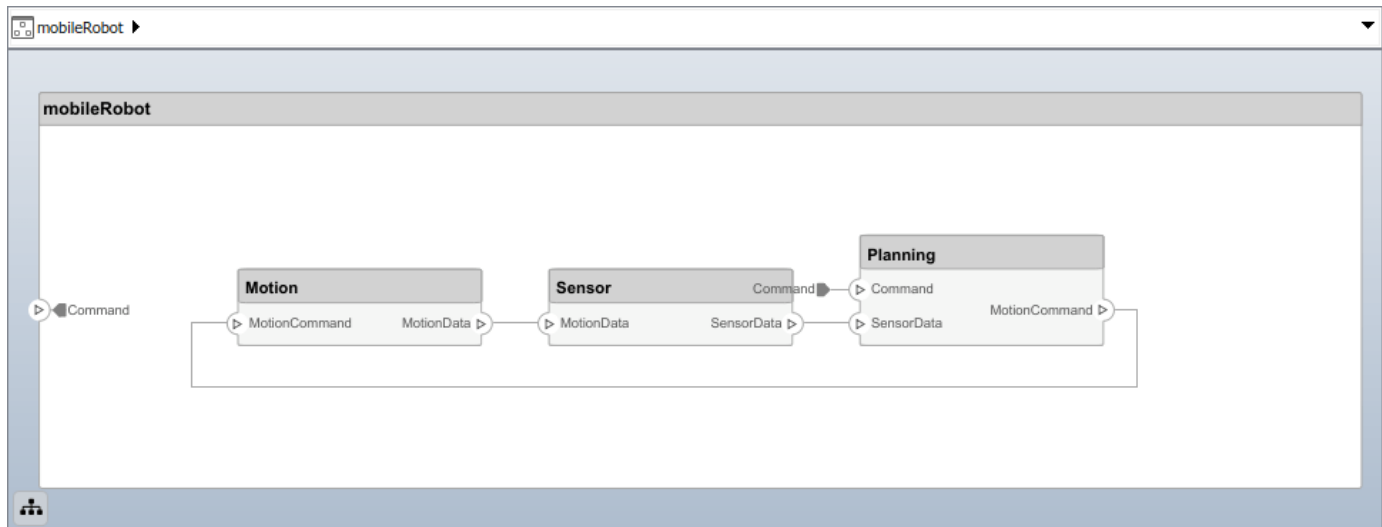
```
model.save
```

Open the model.

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

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. 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 s");
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 contr");
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 interface 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');
```



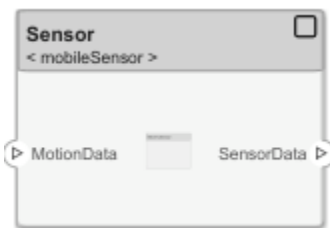
Create Model Reference

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 Sensor component into a reference component to reference the new model. To add additional ports on the Sensor component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "ElectricSensor");
linkDictionary(referenceModel, "SensorInterfaces.sldd");
referenceModel.save
```

```
linkToModel(componentSensor, "mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

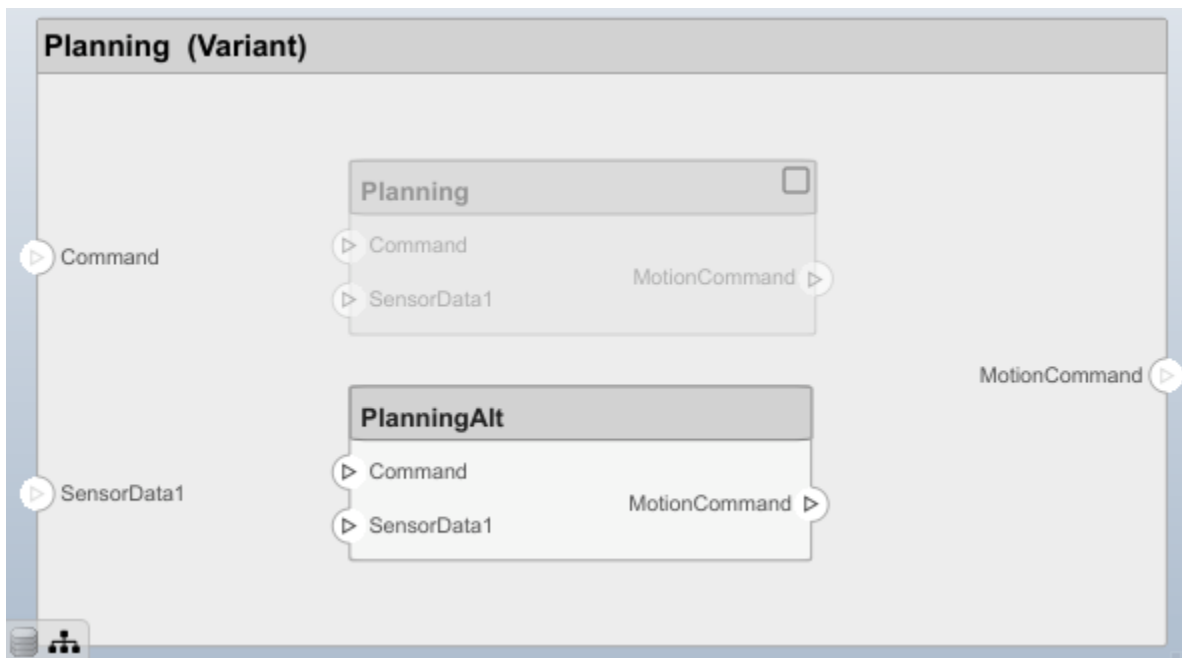
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')
% bdclose('mobileSensor')
% Simulink.data.dictionary.closeAll
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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”

See Also

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

Topics

“Create an Architecture Model”
 “Describe Component Behavior Using Simscape”

Introduced in R2021b

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'

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

Interface associated with port, specified as a `systemcomposer.interface.DataInterface` or `systemcomposer.ValueType` 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 of port, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: double

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of port, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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 component
<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

removeStereotype	Remove stereotype from model 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, and value types, then add components, ports, and connections. 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. 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 S');
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)

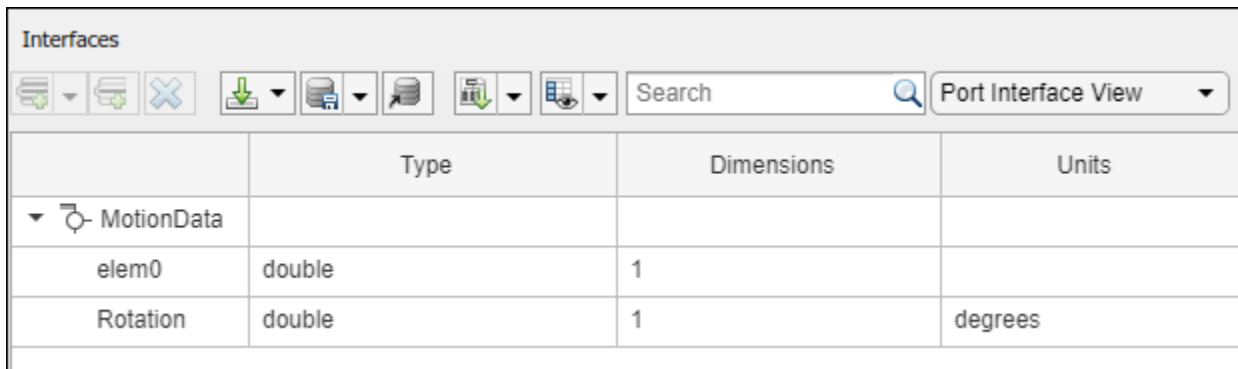
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)

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', "interfaces");
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.

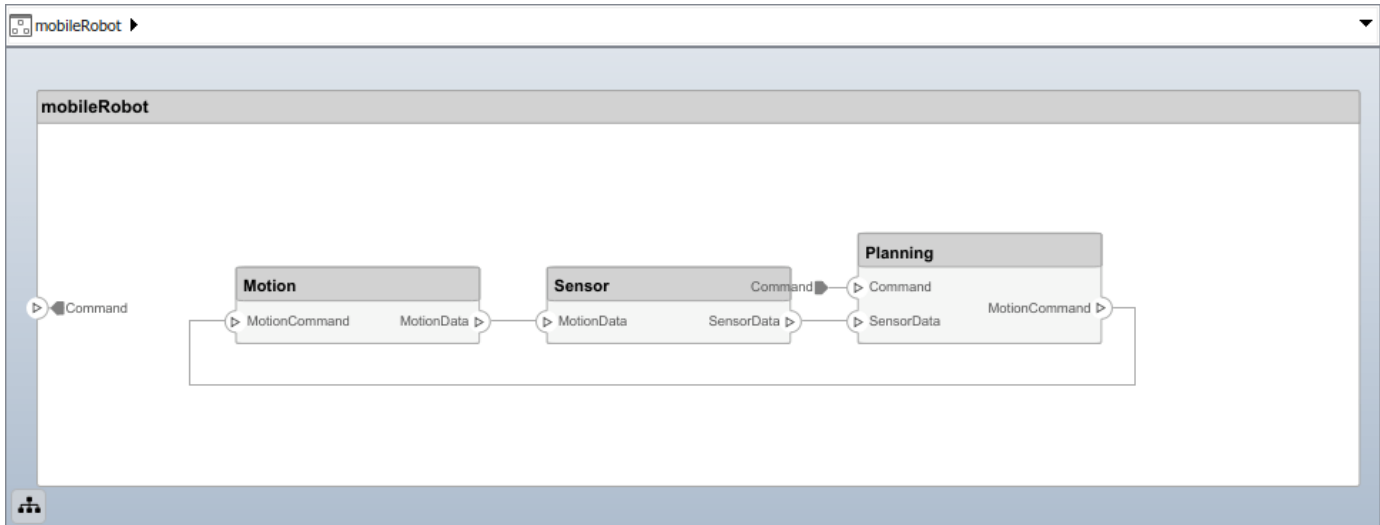
```
model.save
```

Open the model.

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

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. 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");
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 control");
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 interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn')

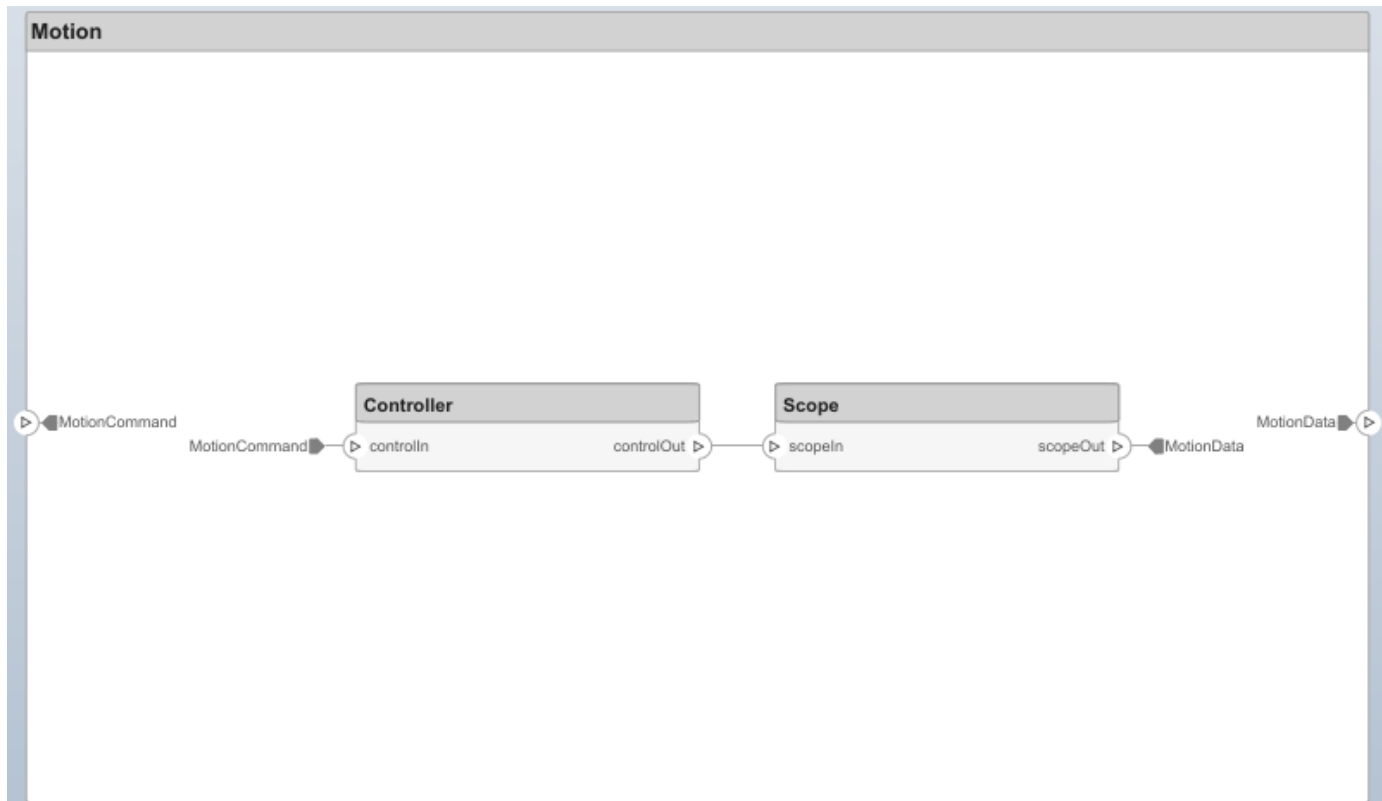
```

Save the model.

```
model.save
```

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

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



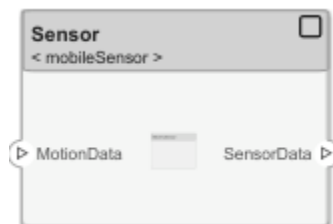
Create Model Reference

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 **Sensor** component into a reference component to reference the new model. To add additional ports on the **Sensor** component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

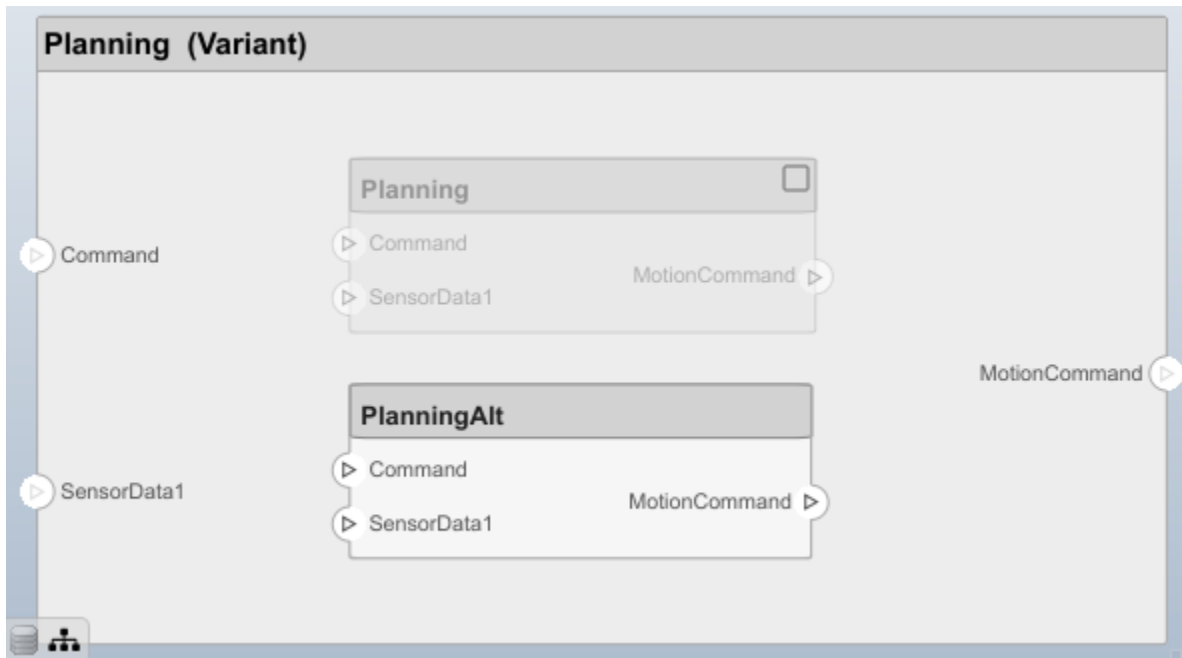
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

systemcomposer.arch.Element | systemcomposer.arch.ComponentPort | systemcomposer.arch.ArchitecturePort | Component

Topics

"Create an Architecture Model"

Introduced in R2019a

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.

```
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.

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 1 (true) or 0 (false).

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 of component, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: double

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of component, specified as a double. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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>createSubsystemBehavior</code>	Add subsystem behavior to component
<code>createStateflowChartBehavior</code>	Add Stateflow chart behavior to component
<code>linkToModel</code>	Link component to a 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 referenced 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 component
<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>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, and value types, then add components, ports, and connections. 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;
```

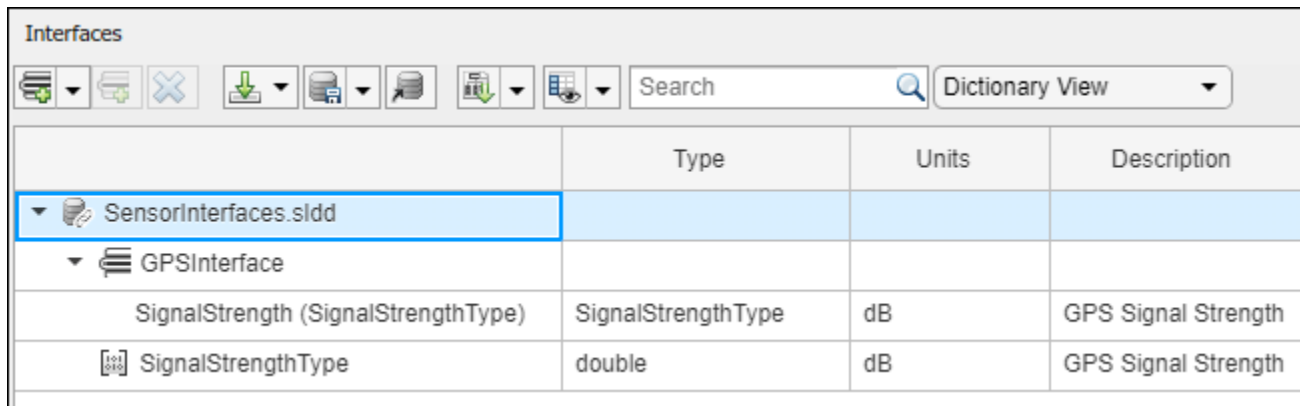
Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Link the data dictionary to the model.

```
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);
linkDictionary(model, "SensorInterfaces.sidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



The screenshot shows the 'Interfaces' editor window. At the top, there are several icons for file operations and a search bar. Below the search bar is a dropdown menu set to 'Dictionary View'. The main area contains a table with the following structure:

	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```


```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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',"interfaces");
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.

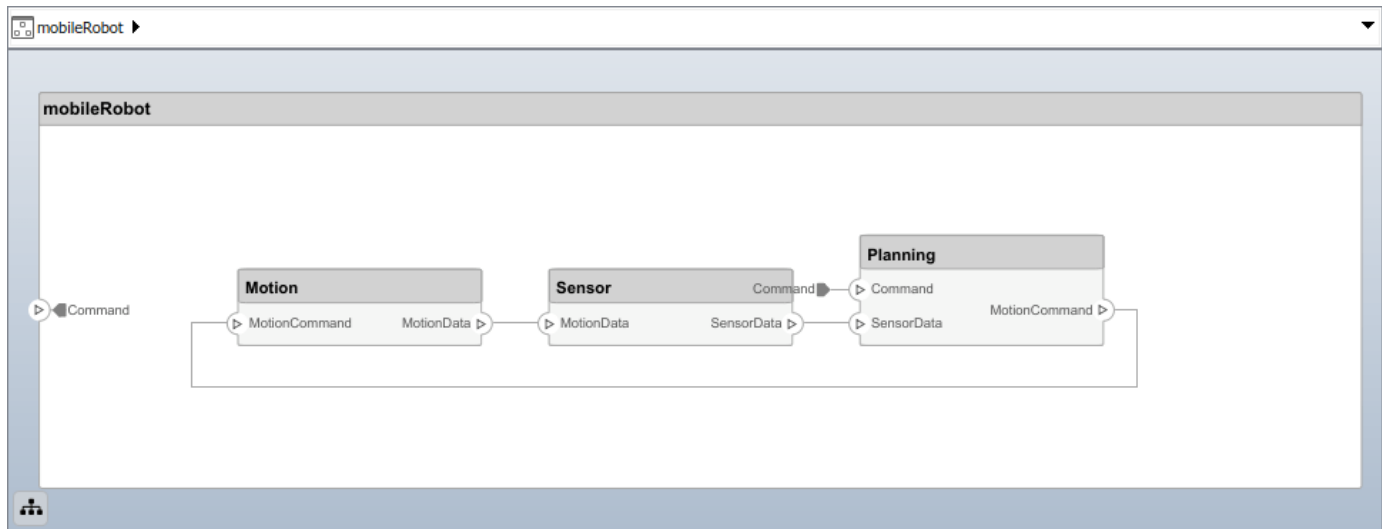
```
model.save
```

Open the model.

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

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. 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 ');
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 control');
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 interface 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');
```



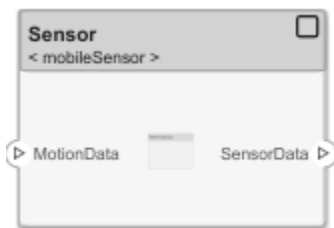
Create Model Reference

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 Sensor component into a reference component to reference the new model. To add additional ports on the Sensor component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "ElectricSensor");
linkDictionary(referenceModel, "SensorInterfaces.sldd");
referenceModel.save
```

```
linkToModel(componentSensor, "mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

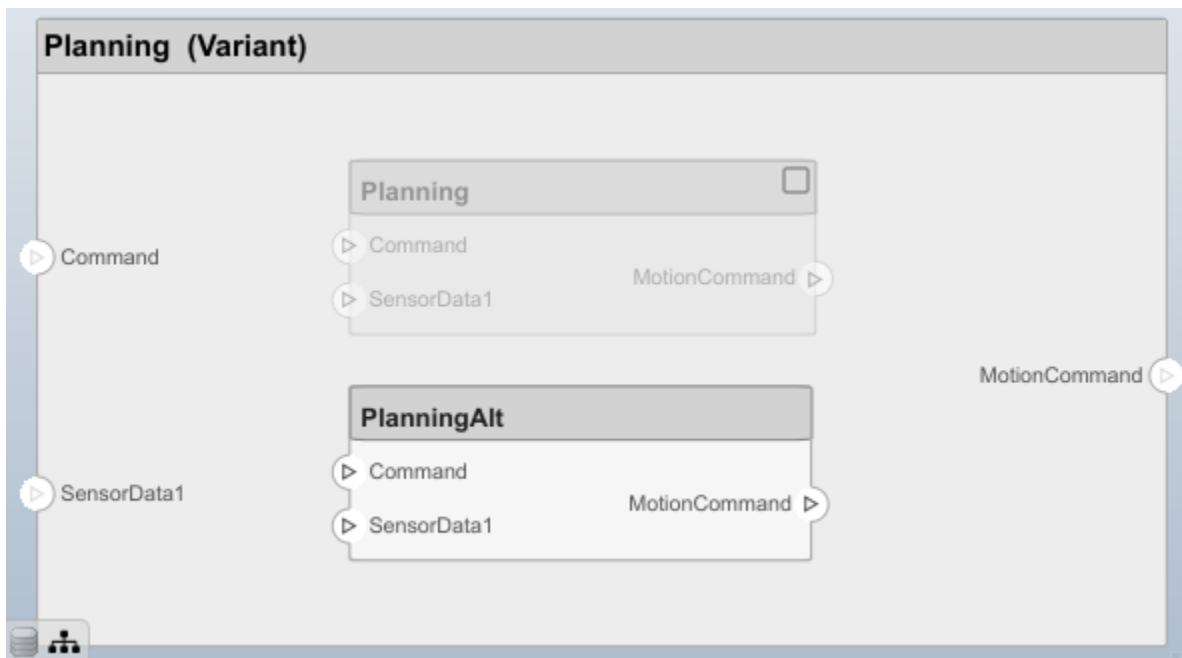
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')
% bdclose('mobileSensor')
% Simulink.data.dictionary.closeAll
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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”

See Also

systemcomposer.arch.Architecture | systemcomposer.arch.Element | createModel | addComponent | Component

Topics

“Create an Architecture Model”

Introduced in R2019a

systemcomposer.arch.ComponentPort

Input and output ports of component

Description

A ComponentPort object represents the input and output 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 and then getting the component port.

```
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'

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

Interface associated with port, specified as a `systemcomposer.interface.DataInterface` or `systemcomposer.ValueType` 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 of port, specified as a double. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: double

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of port, specified as a double. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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 component
<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, and value types, then add components, ports, and connections. 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. 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 S');
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

Interfaces			
Search			
Dictionary View			
	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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			
Search			
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',"interfaces");
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.

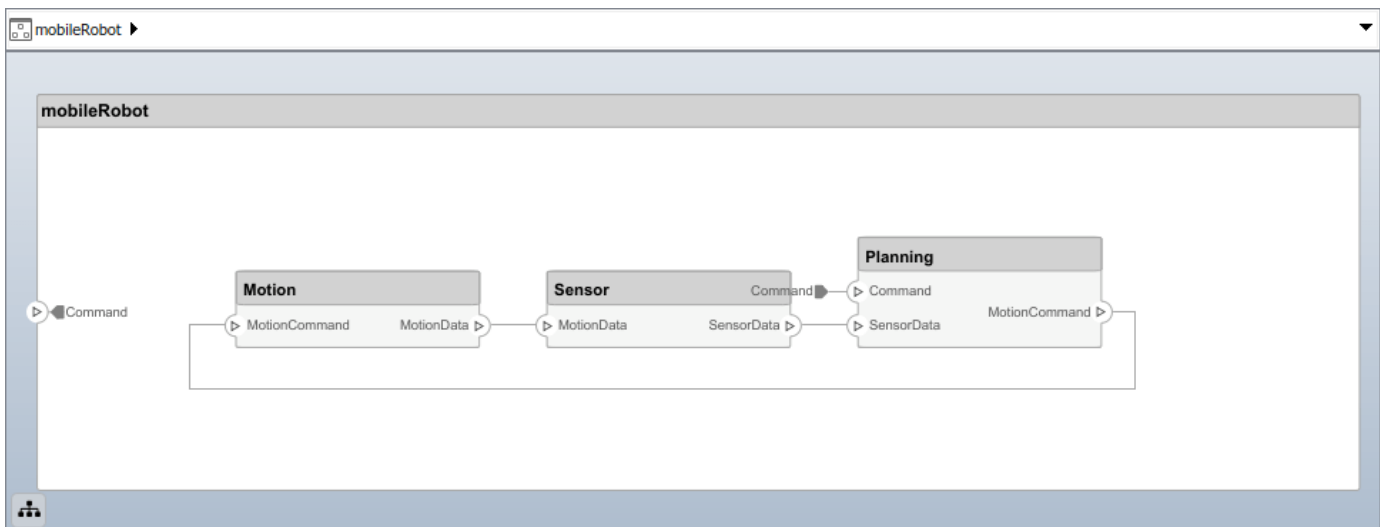
```
model.save
```

Open the model.

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

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. 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");
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 control'");
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');

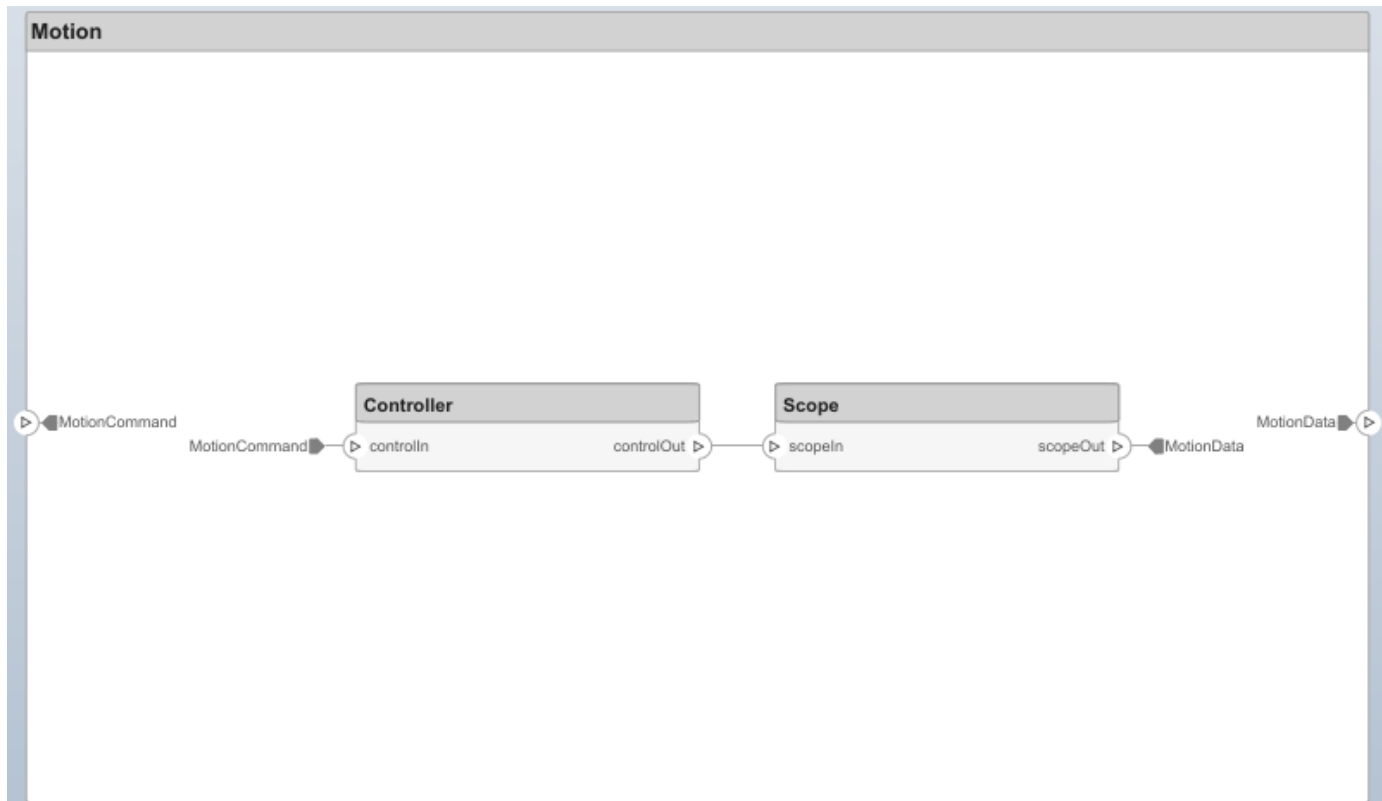
c_planningController = connect(motionPorts(1), controllerCompPortIn);
% For outport connections, the interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn');
```

Save the model.

```
model.save
```

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

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



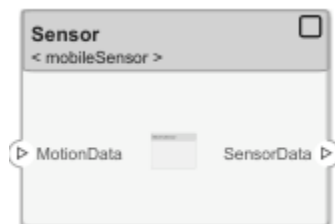
Create Model Reference

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 `Sensor` component into a reference component to reference the new model. To add additional ports on the `Sensor` component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

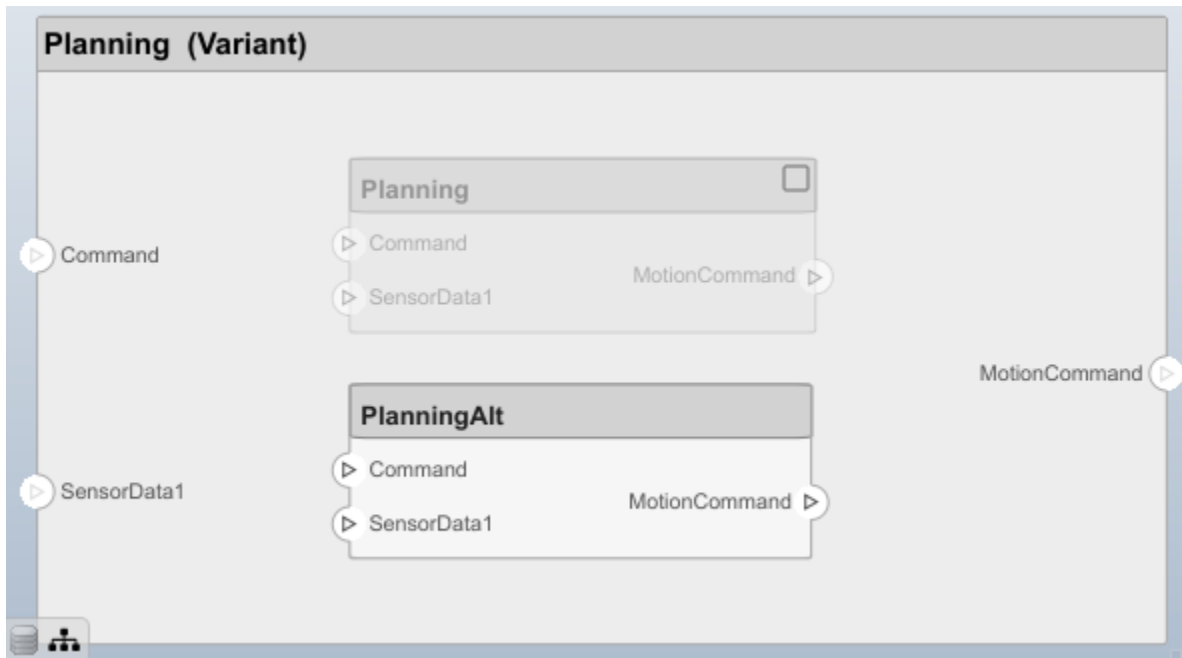
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

systemcomposer.arch.ArchitecturePort | systemcomposer.arch.BasePort | systemcomposer.arch.Element | getPort | addPort | Component

Topics

"Create an Architecture Model"

Introduced in R2019a

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.

```
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

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 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 of connector, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: `double`

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of connector, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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 component
<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, and value types, then add components, ports, and connections. 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;
```

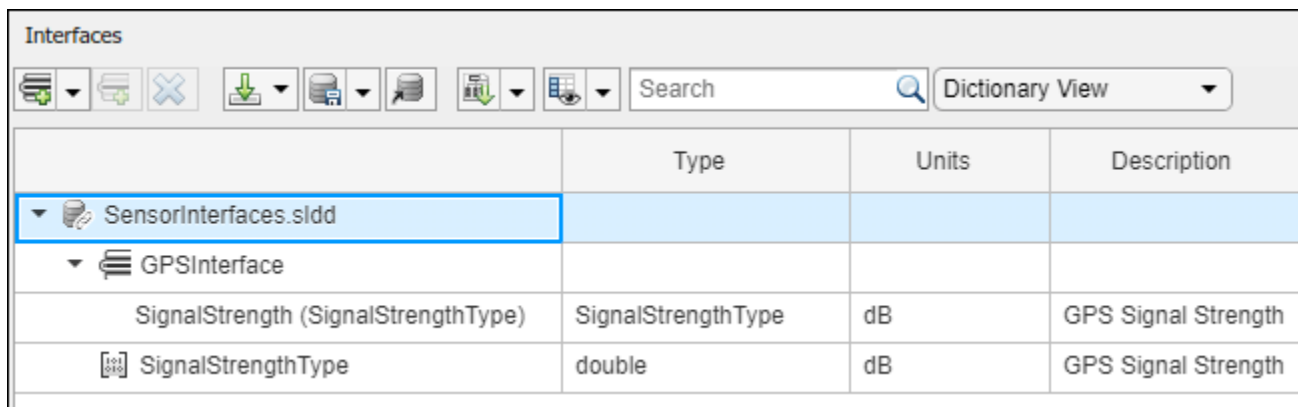
Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Link the data dictionary to the model.

```
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);
linkDictionary(model, "SensorInterfaces.sidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



The screenshot shows the 'Interfaces' window in System Composer. It features a toolbar with icons for creating, deleting, and linking dictionaries and interfaces, a search bar, and a 'Dictionary View' dropdown. The main area contains a table with the following data:

	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

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

```
sensorPorts(2).setInterface(interface)
```

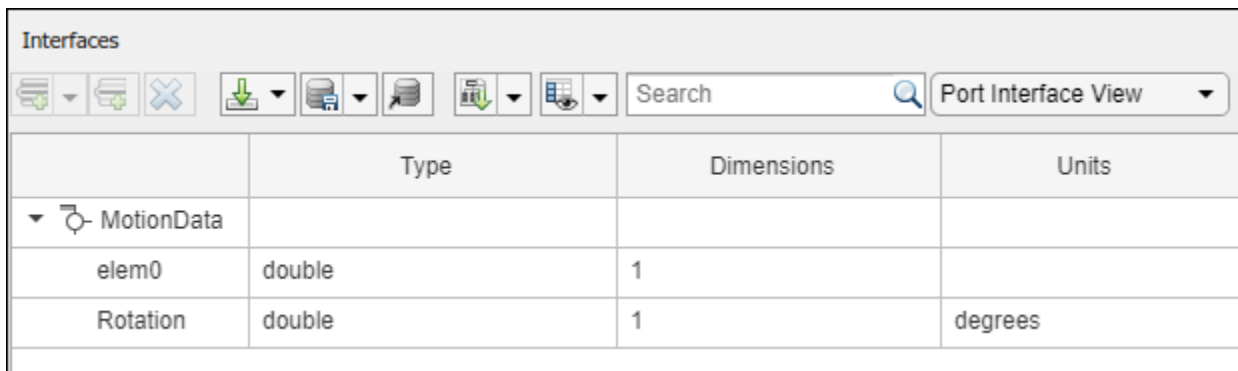
```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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', "interfaces");
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.

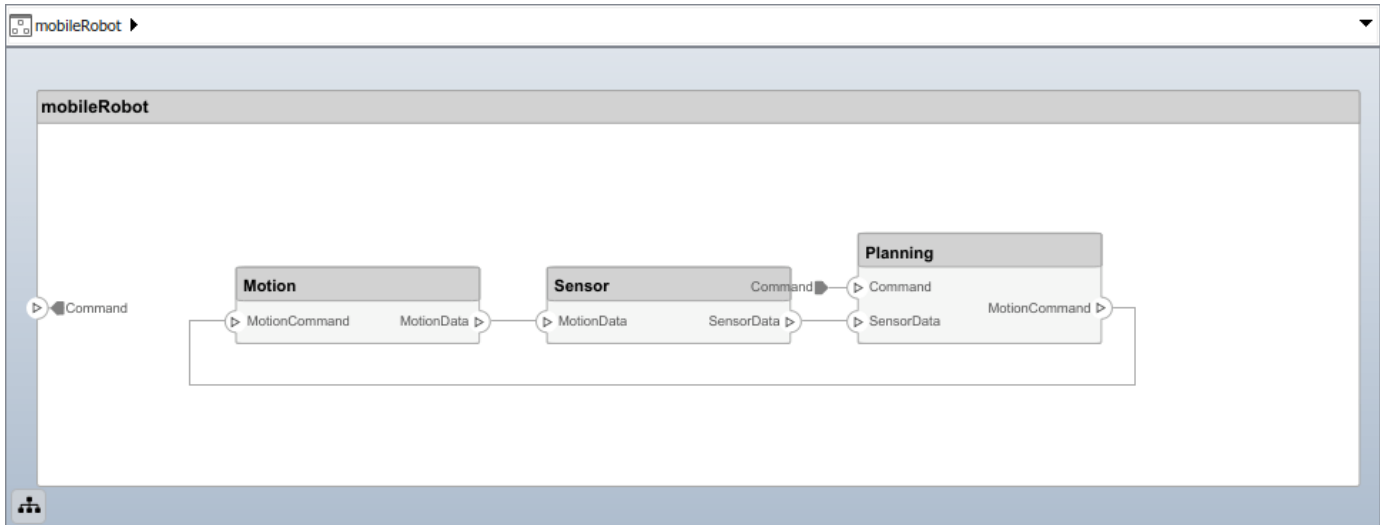
```
model.save
```

Open the model.


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

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. 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");
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 control");
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 interface element must be specified
c_planningScope = connect(scopeCompPortOut,motionPorts(2),'DestinationElement',"Rotation");
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')

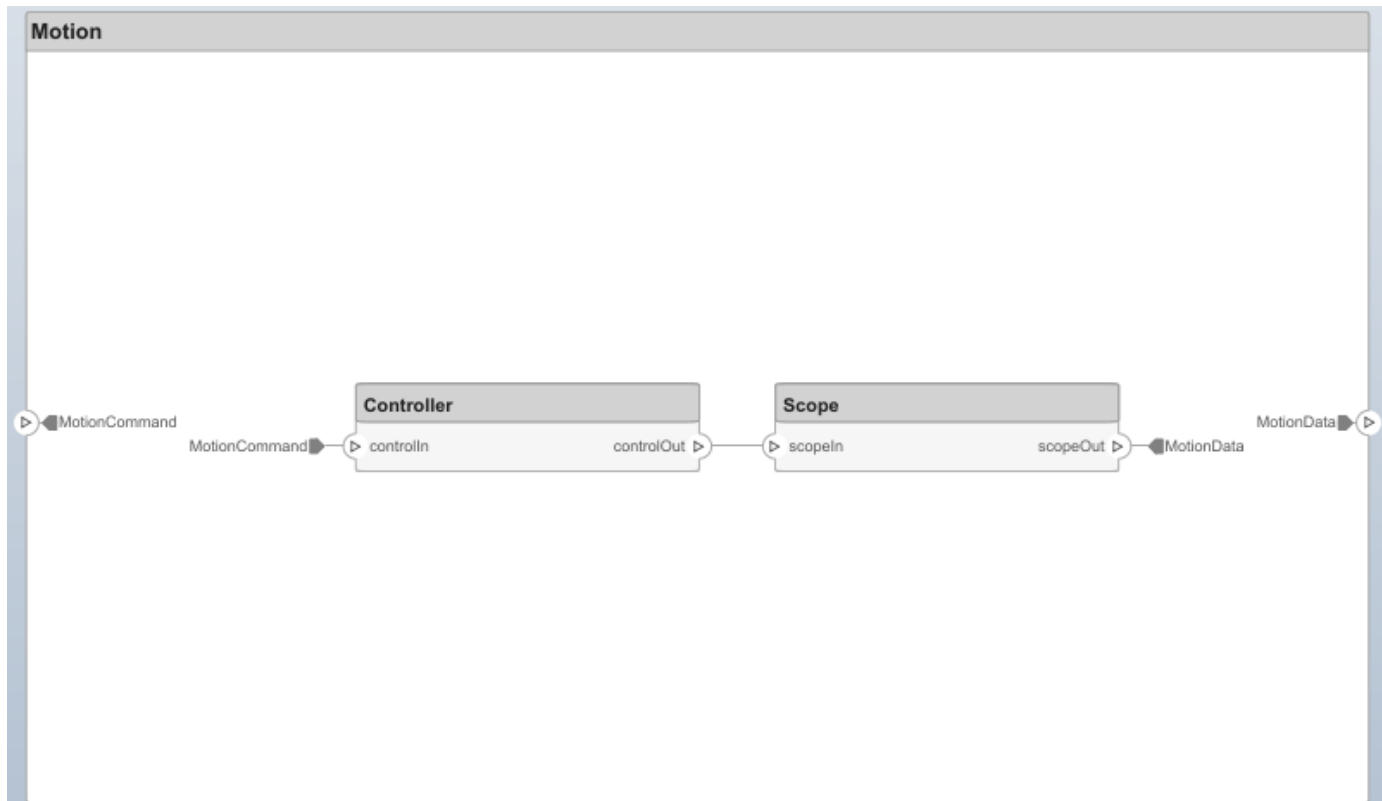
```

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.

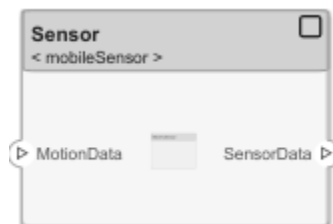
Create a new System Composer model. Convert the **Sensor** component into a reference component to reference the new model. To add additional ports on the **Sensor** component, you must update the referenced model "mobileSensor".

```

referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
  
```

```

linkToModel(componentSensor,"mobileSensor");
  
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

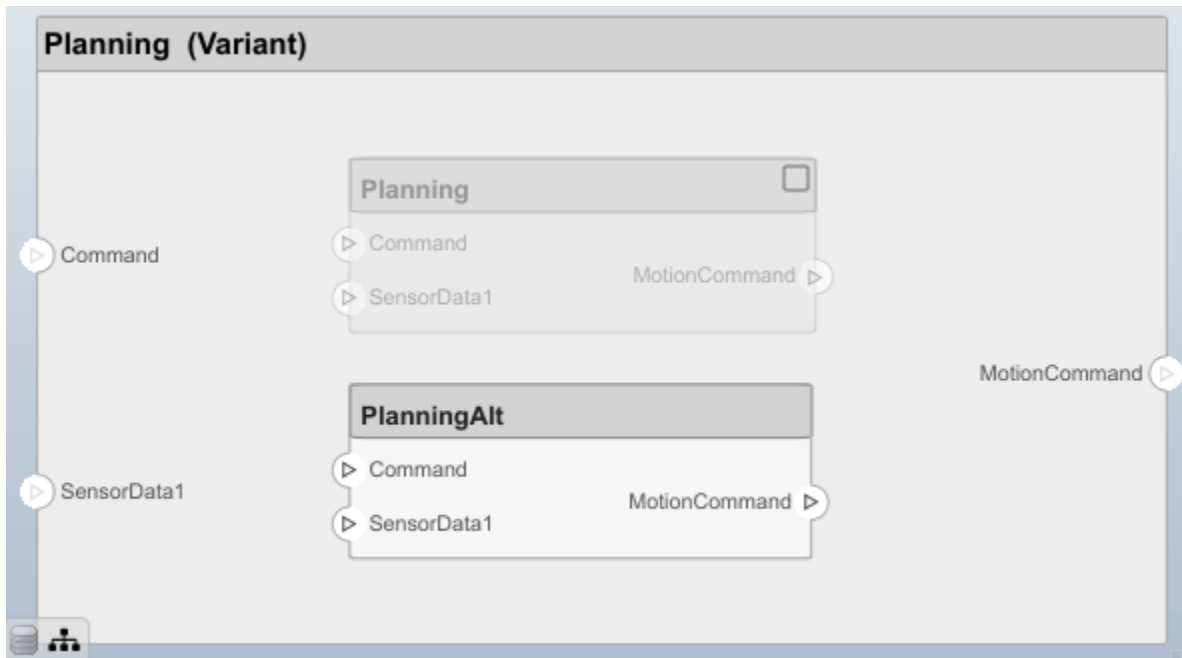
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

systemcomposer.arch.Element | systemcomposer.arch.BaseConnector | systemcomposer.arch.PhysicalConnector | connect | Component

Topics

"Create an Architecture Model"

Introduced in R2019a

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, port, or connector: `addComponent`, `addPort`, `connect`.

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 of element, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: `double`

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of element, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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 component
<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, and value types, then add components, ports, and connections. 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;
```

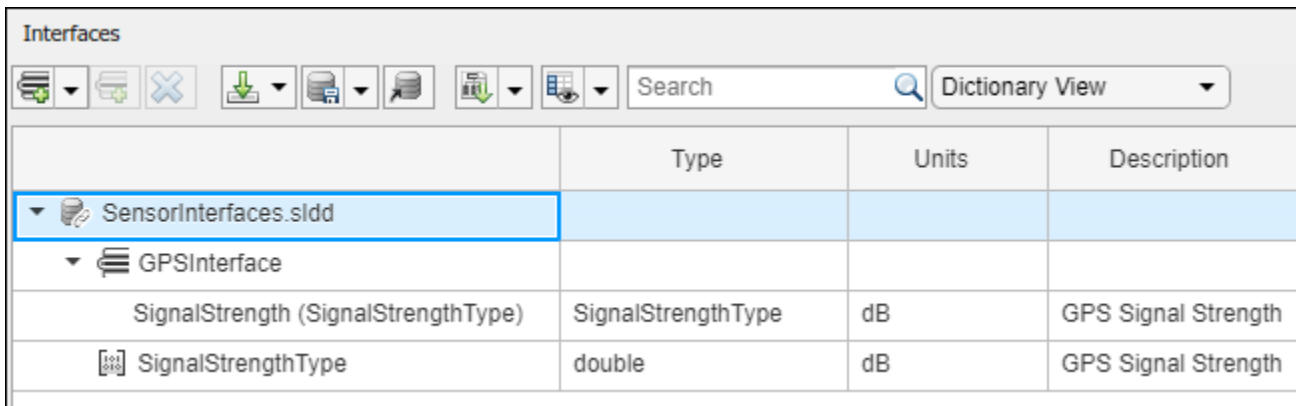
Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Link the data dictionary to the model.

```
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);
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)




componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)

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"/>  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','interfaces');
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.

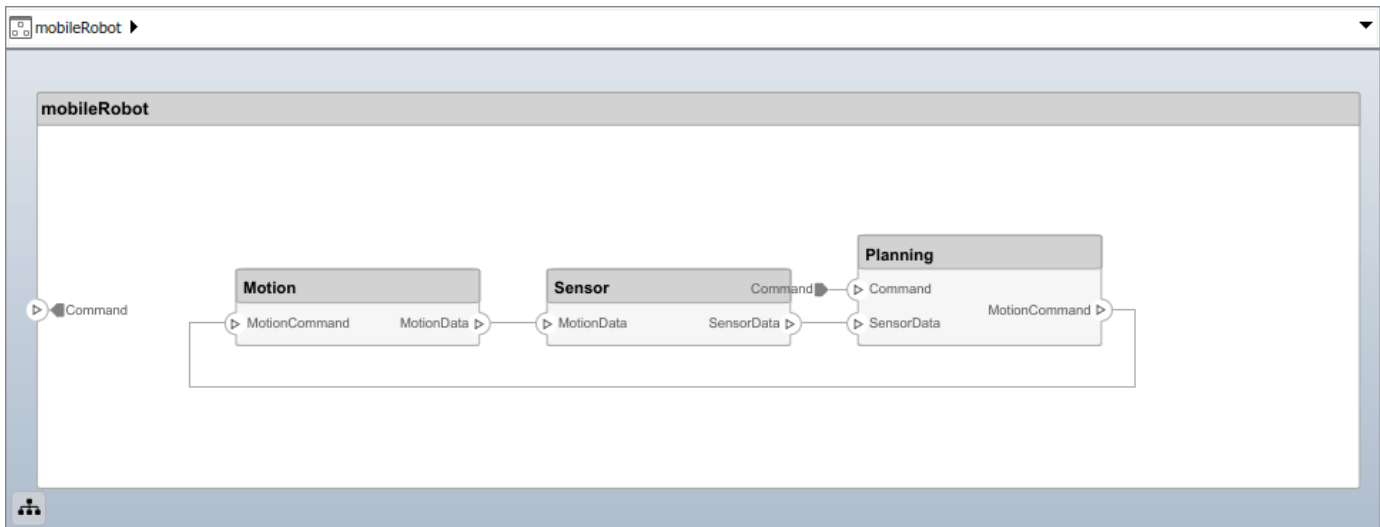
```
model.save
```

Open the model.

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

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. 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 s");
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 contr");
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 interface 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');
```



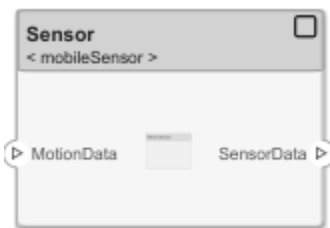
Create Model Reference

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 Sensor component into a reference component to reference the new model. To add additional ports on the Sensor component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "ElectricSensor");
linkDictionary(referenceModel, "SensorInterfaces.sldd");
referenceModel.save
```

```
linkToModel(componentSensor, "mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```


Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

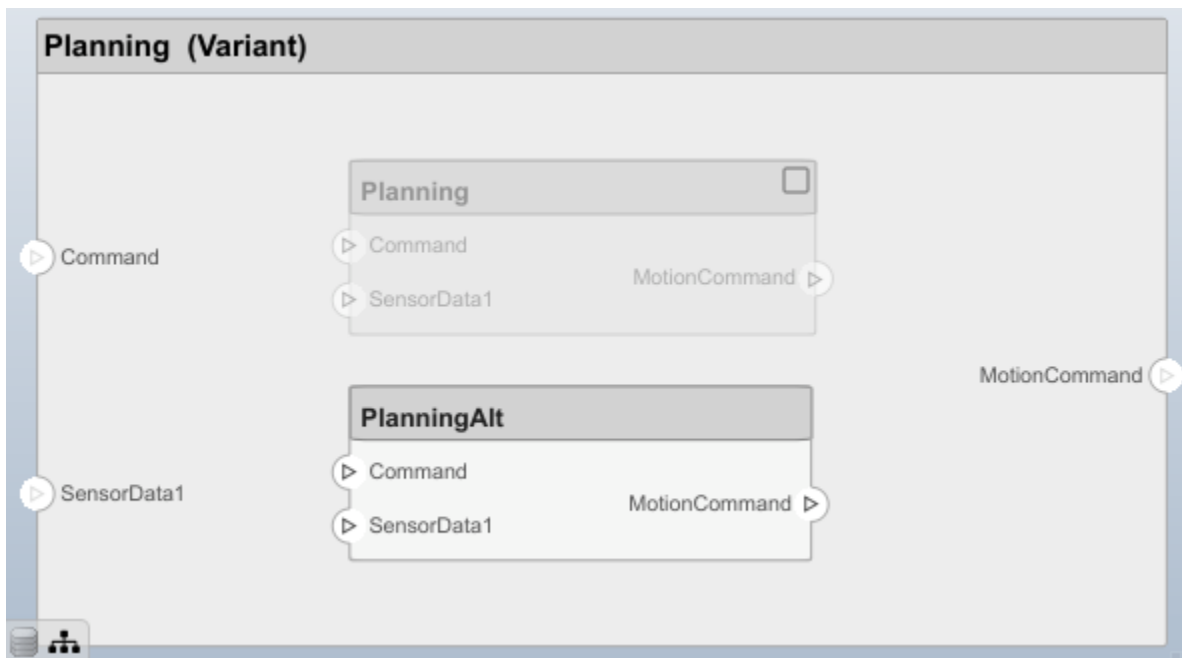
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')
% bdclose('mobileSensor')
% Simulink.data.dictionary.closeAll
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

Topics

"Create an Architecture Model"

Introduced in R2019a

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.

```
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.

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`

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.

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

`increaseExecutionOrder` Change function execution order to later
`decreaseExecutionOrder` Change function execution order to earlier

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"> “Author Software Architectures” “Simulate and Deploy Software Architectures”
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"> “Implement Behaviors for Architecture Model Simulation” “Create Software Architecture from Component”
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 the Software Architecture of a Throttle Position Control System”
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”

See Also

`systemcomposer.createModel` | `createArchitectureModel` | `createSimulinkBehavior`

Topics

“Modeling the Software Architecture of a Throttle Position Control System”

“Simulate and Deploy Software Architectures”

“Author Software Architectures”

Introduced in R2021b

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.

```
objModel = systemcomposer.createModel('NewModel')
```

The createModel method is the constructor for the systemcomposer.arch.Model class.

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 numeric value.

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 model
<code>save</code>	Save architecture model or data dictionary
<code>find</code>	Find architecture model elements using query
<code>lookup</code>	Search for architecture element
<code>openViews</code>	Open architecture views editor
<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, and value types, then add components, ports, and connections. 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. Link the data dictionary to the model.

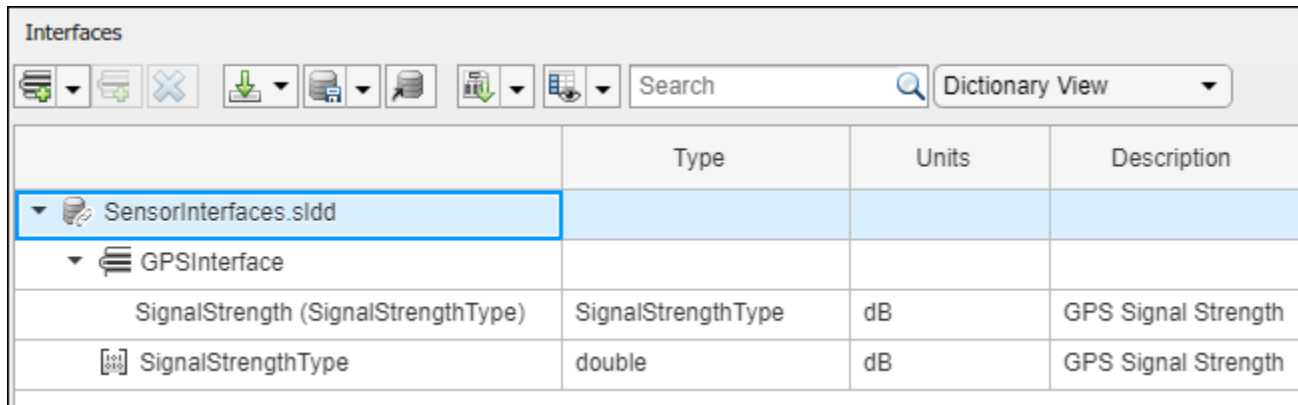
```
dictionary = systemcomposer.createDictionary("SensorInterfaces.sldd");
interface = dictionary.addInterface("GPSInterface");
element = interface.addElement("SignalStrength");
valueType = dictionary.addValueType("SignalStrengthType", 'Units', 'dB', 'Description', 'GPS Signal');
```

```
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.sidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



The screenshot shows the 'Interfaces' editor window. At the top, there is a toolbar with various icons and a search box. Below the toolbar is a table with the following columns: Type, Units, and Description. The table content is as follows:

	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```



```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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"/> 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','interfaces');
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.

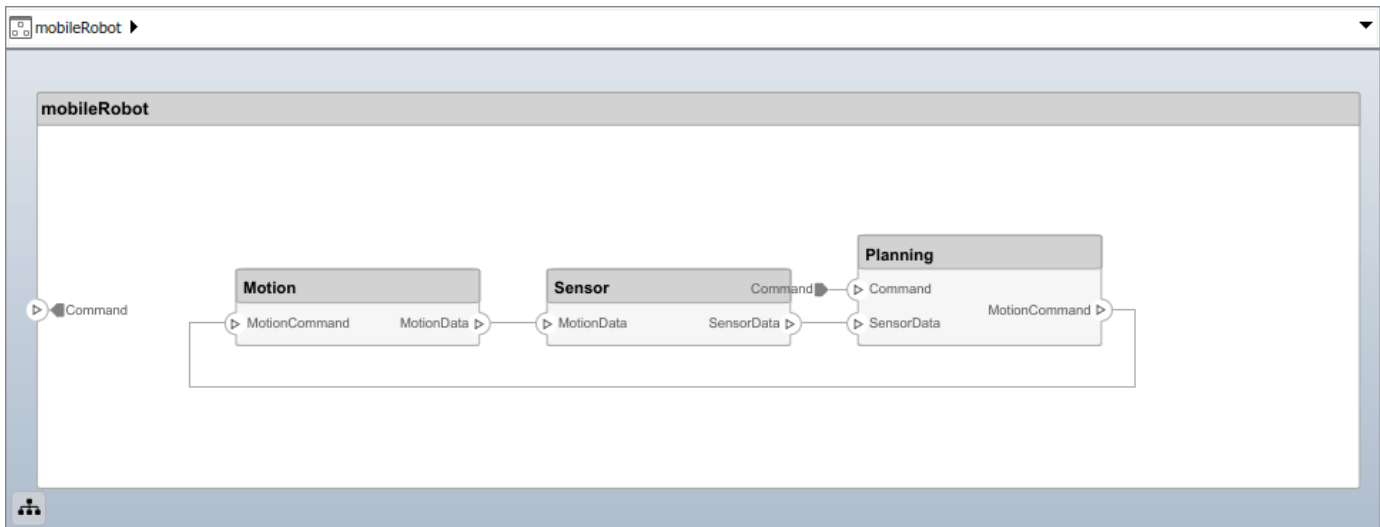
```
model.save
```

Open the model.

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

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. 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 s");
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 contr");
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 interface 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');
```



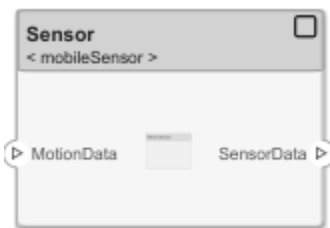
Create Model Reference

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 Sensor component into a reference component to reference the new model. To add additional ports on the Sensor component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "ElectricSensor");
linkDictionary(referenceModel, "SensorInterfaces.sldd");
referenceModel.save
```

```
linkToModel(componentSensor, "mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```


Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

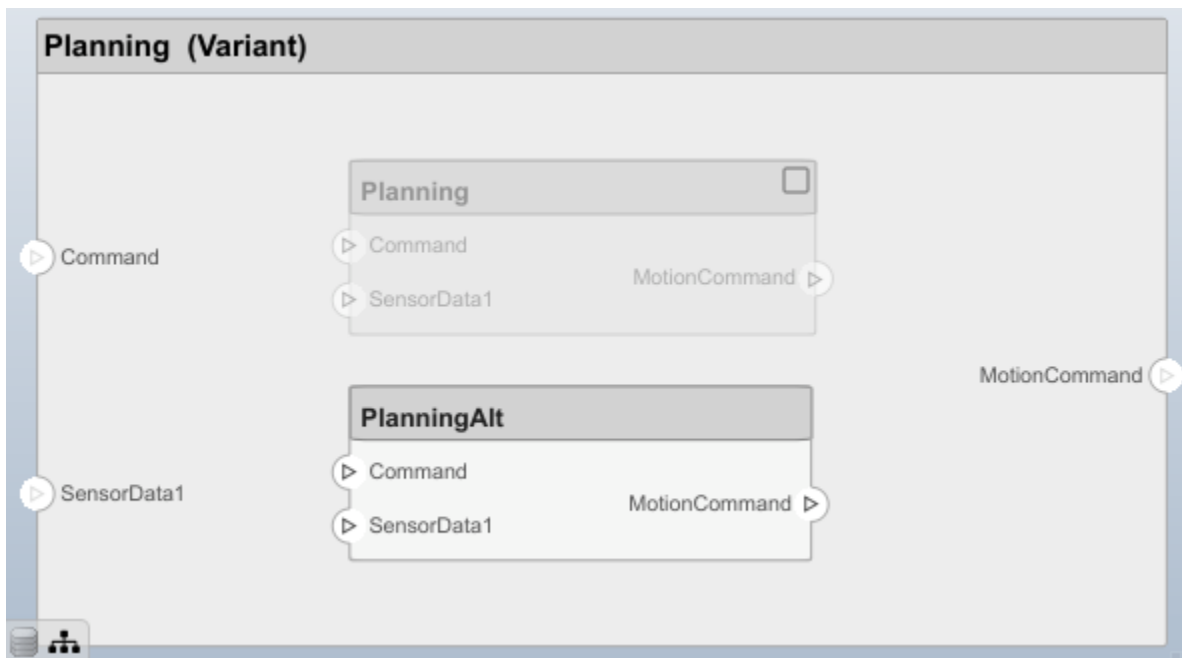
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')
% bdclose('mobileSensor')
% Simulink.data.dictionary.closeAll
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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"

See Also

createModel | loadModel | importModel | exportModel | openModel | createArchitectureModel

Topics

"Create an Architecture Model"

Introduced in R2019a

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.

```
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 of connector, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: double

SimulinkModelHandle – Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of connector, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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 component
<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

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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”

See Also

`systemcomposer.arch.Element` | `systemcomposer.arch.BaseConnector` | `systemcomposer.arch.Connector` | `connect` | `Component`

Topics

“Create an Architecture Model”
 “Describe Component Behavior Using Simscape”

Introduced in R2021b

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.

```
varComp = addVariantComponent(archObj, 'compName');
```

The `addVariantComponent` method creates a `systemcomposer.arch.VariantComponent` object.

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.

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 of component, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: `double`

SimulinkModelHandle — Simulink handle to parent model

numeric value

Simulink handle to parent System Composer model of component, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirements APIs.

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

Data Types: `double`

Object Functions

addChoice	Add variant choices to variant component
setCondition	Set condition on variant choice
setActiveChoice	Set active choice on variant component
getChoices	Get available choices in variant component
getActiveChoice	Get active choice on variant component
getCondition	Return variant control on choice within variant component
applyStereotype	Apply stereotype to architecture model element
getStereotypes	Get stereotypes applied on element of architecture model
removeStereotype	Remove stereotype from model element
getPort	Get port from component
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from component
getStereotypeProperties	Get stereotype property names on element
getProperty	Get property value corresponding to stereotype applied to element
setProperty	Set property value corresponding to stereotype applied to 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, and value types, then add components, ports, and connections. 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;
```

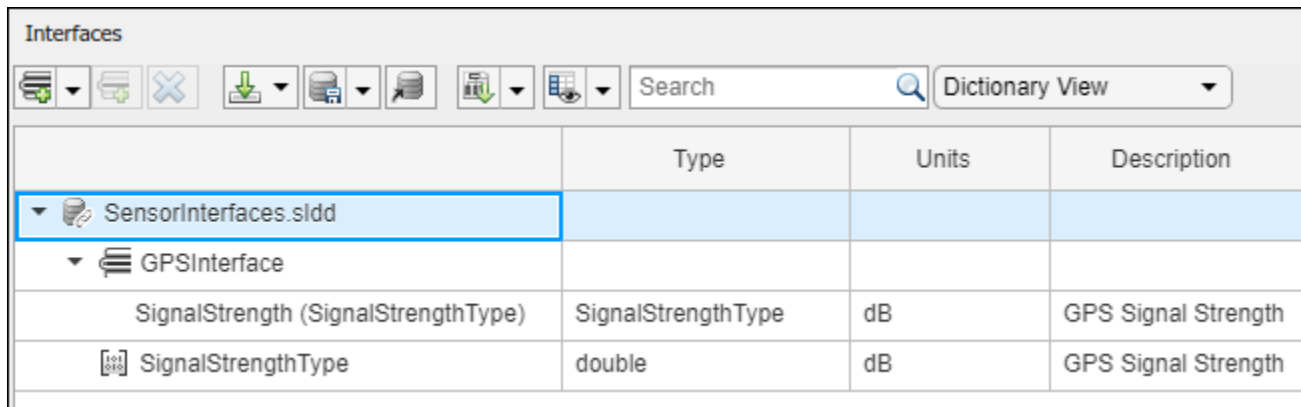
Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Link the data dictionary to the model.

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

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



Interfaces			
Search			
Dictionary View			
	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

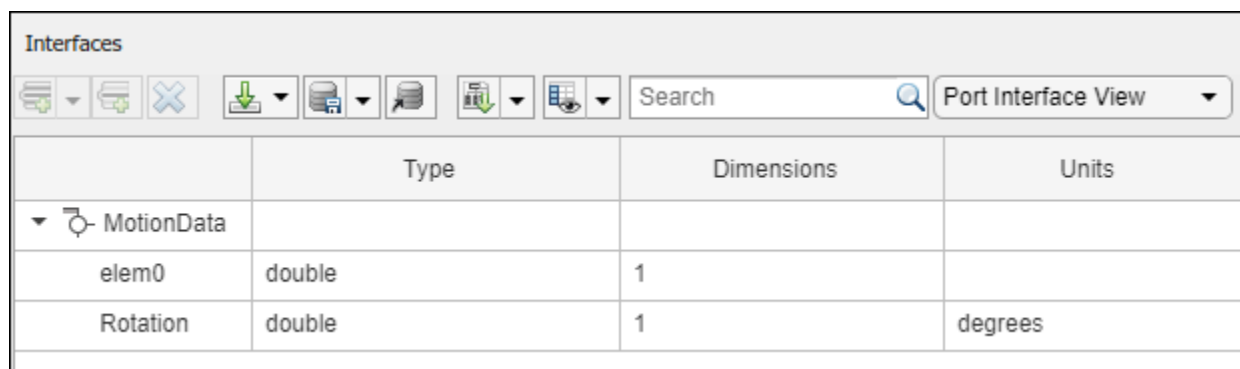
```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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			
Search			
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',"interfaces");
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.

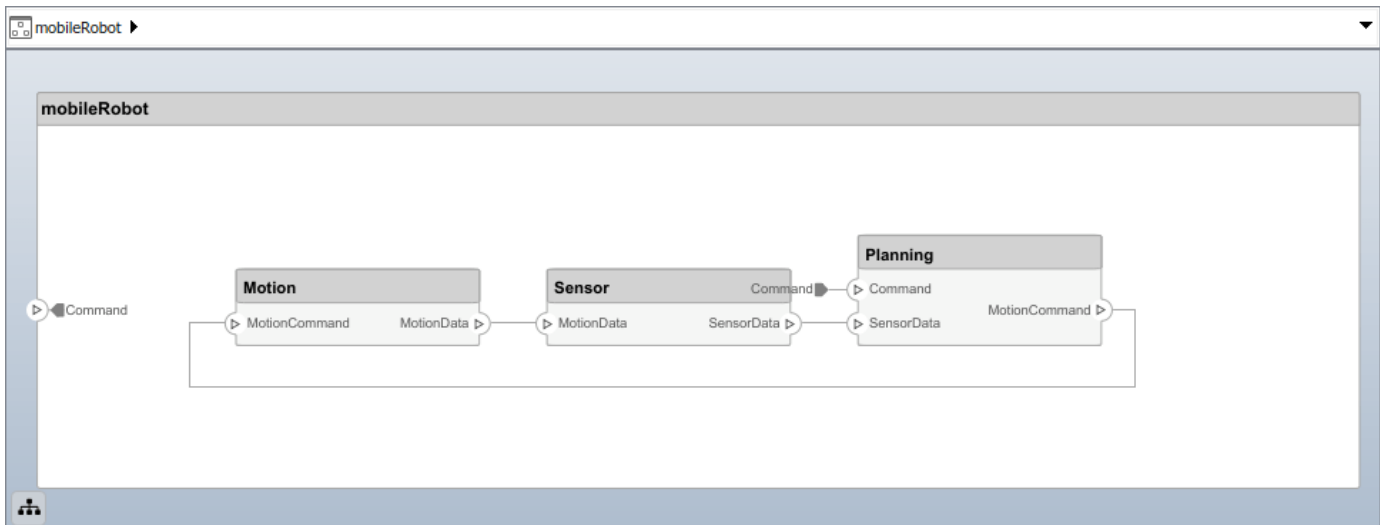
```
model.save
```

Open the model.

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

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. 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");
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 control'");
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');

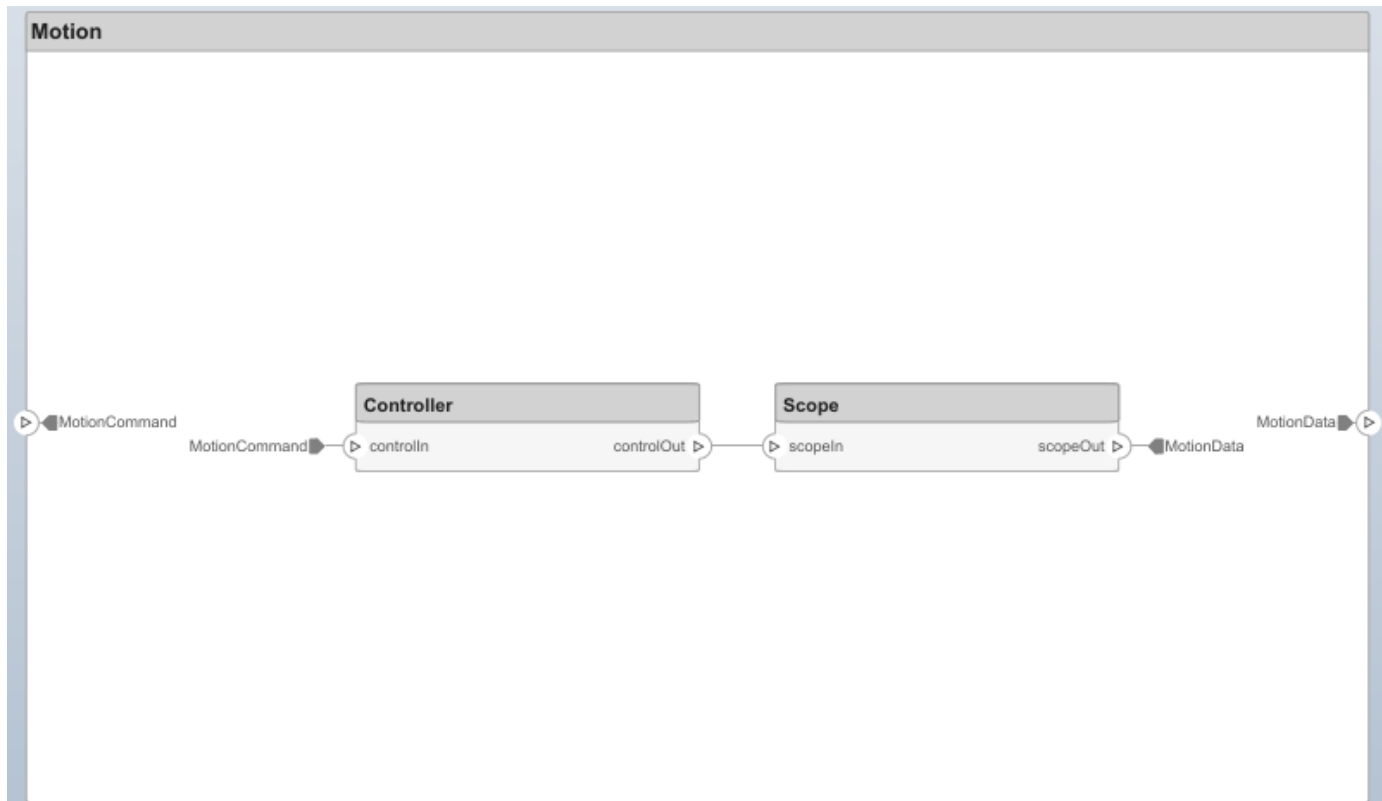
c_planningController = connect(motionPorts(1), controllerCompPortIn);
% For outport connections, the interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn');
```

Save the model.

```
model.save
```

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

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



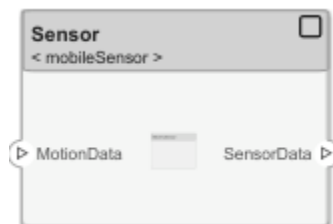
Create Model Reference

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 **Sensor** component into a reference component to reference the new model. To add additional ports on the **Sensor** component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

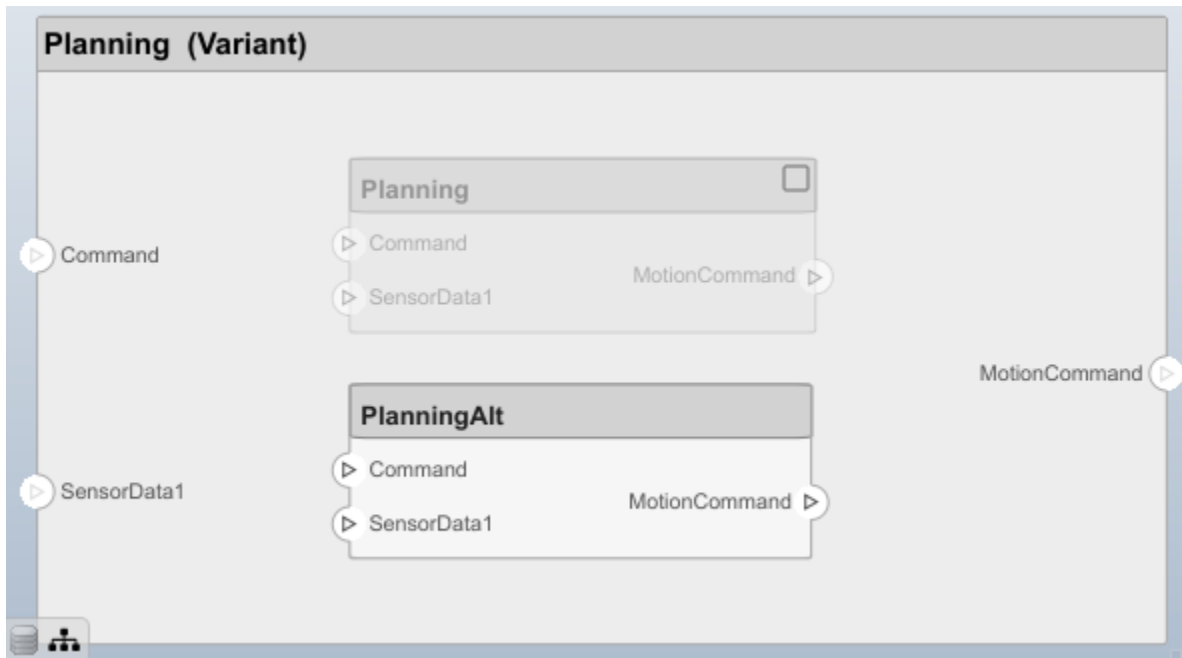
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Condition" on page 1-531

See Also

Variant Component

Topics

"Decompose and Reuse Components"

Introduced in R2019a

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.

```
dictionary = systemcomposer.createDictionary('newDictionary.slidd');
```

Properties

Interfaces — Interfaces defined in dictionary

array of interface objects

Interfaces defined in dictionary, specified as an array of `systemcomposer.interface.DataInterface` or `systemcomposer.interface.PhysicalInterface` 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

<code>addValueType</code>	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>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, and value types, then add components, ports, and connections. 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. 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 S');  
element.setType(valueType);  
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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','interfaces');
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.

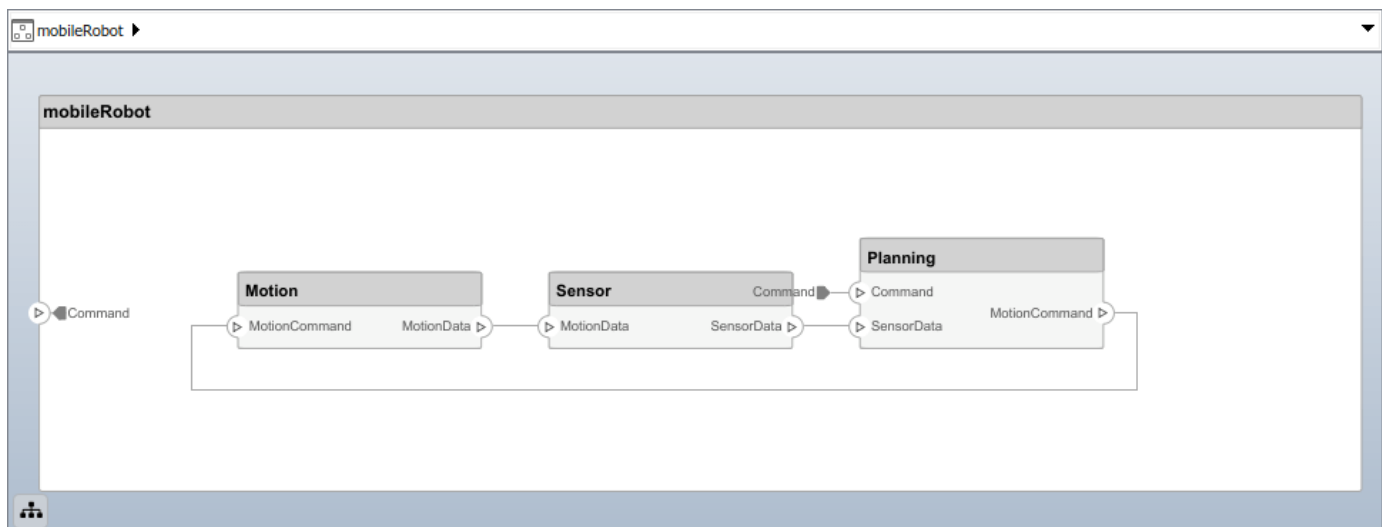
```
model.save
```

Open the model.

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

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. 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');
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 control');
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 interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn');

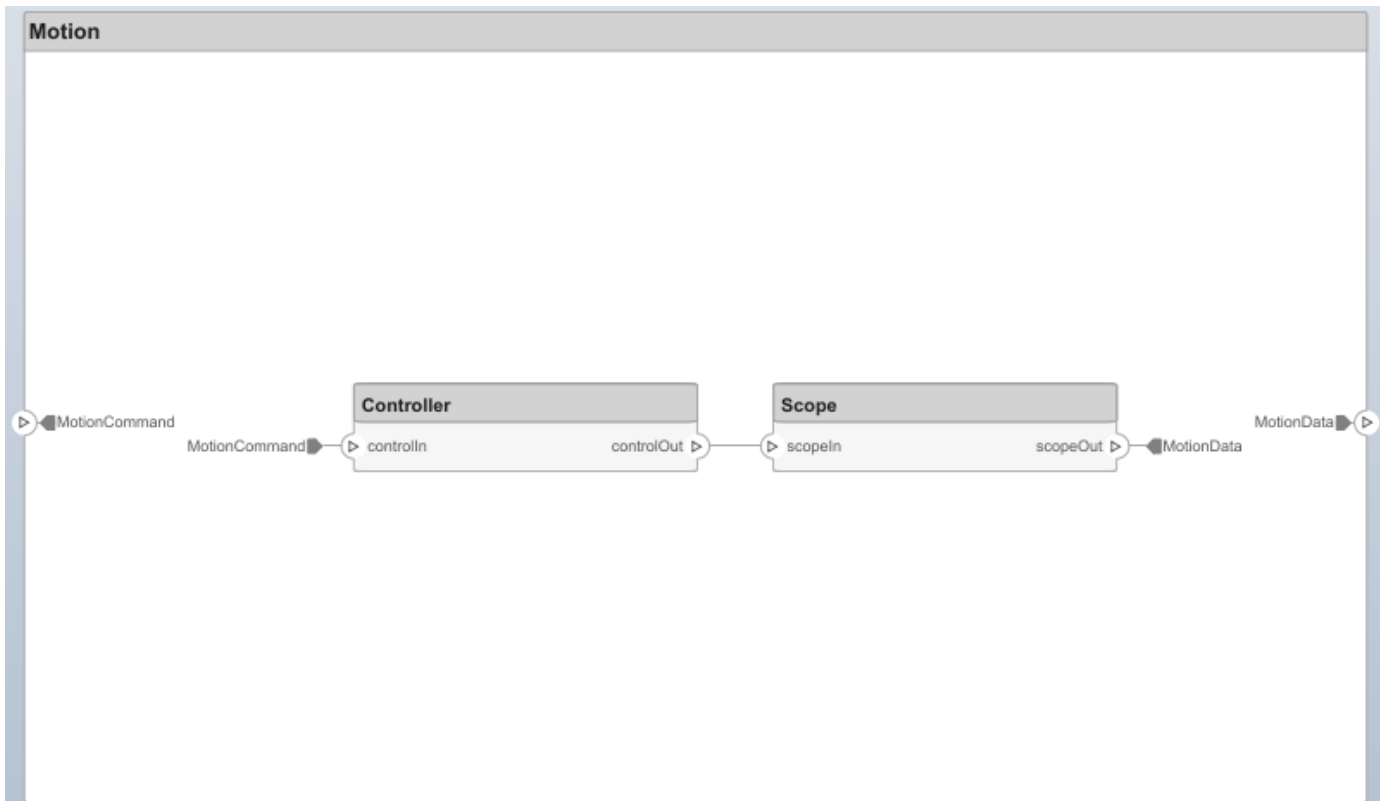
```

Save the model.

```
model.save
```

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

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



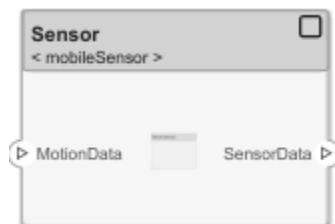
Create Model Reference

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 `Sensor` component into a reference component to reference the new model. To add additional ports on the `Sensor` component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

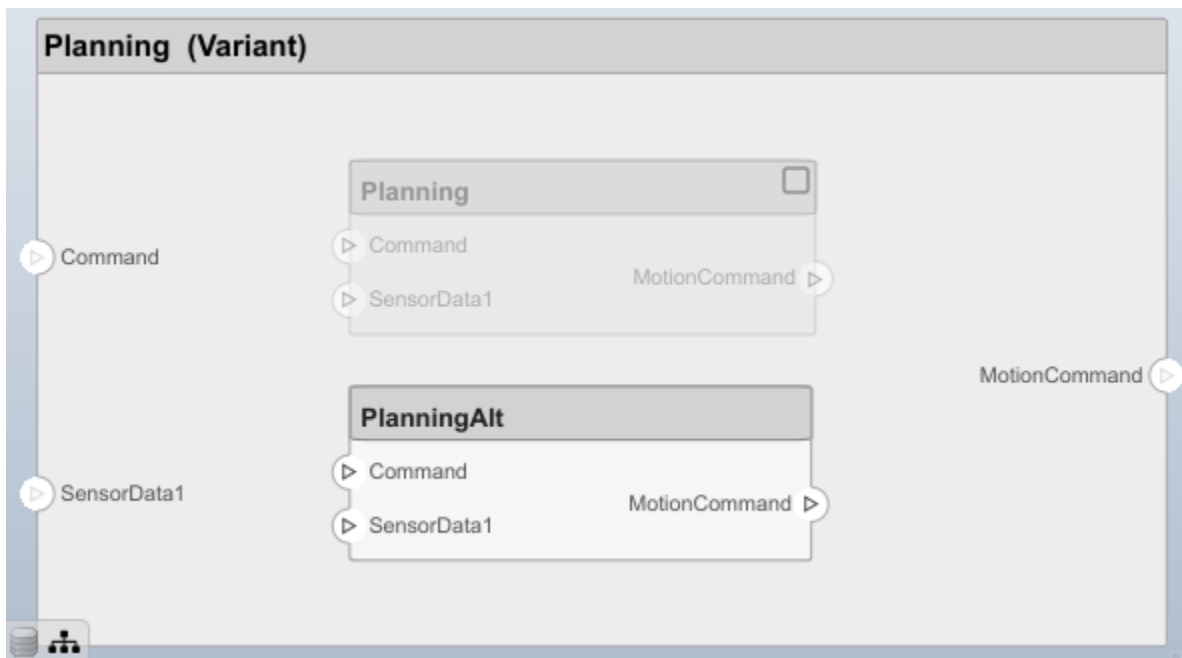
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

[openDictionary](#) | [createDictionary](#) | [saveToDictionary](#) | [systemcomposer.ValueType](#) | [systemcomposer.interface.DataElement](#) | [systemcomposer.interface.DataInterface](#) | [systemcomposer.interface.PhysicalInterface](#) | [systemcomposer.interface.PhysicalElement](#) | [systemcomposer.interface.PhysicalDomain](#)

Topics

["Create Interfaces"](#)
["Manage Interfaces with Data Dictionaries"](#)
["Specify Physical Interfaces on the Ports"](#)

Introduced in R2019a

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, 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
destroy	Remove model element

Compatibility Considerations

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”

Introduced in R2019a

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.

```
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.

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, interface, or element
setType	Set shared type on data element
createOwnedType	Create owned type on data 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, and value types, then add components, ports, and connections. 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. 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 S');
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.slidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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','interfaces');
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.

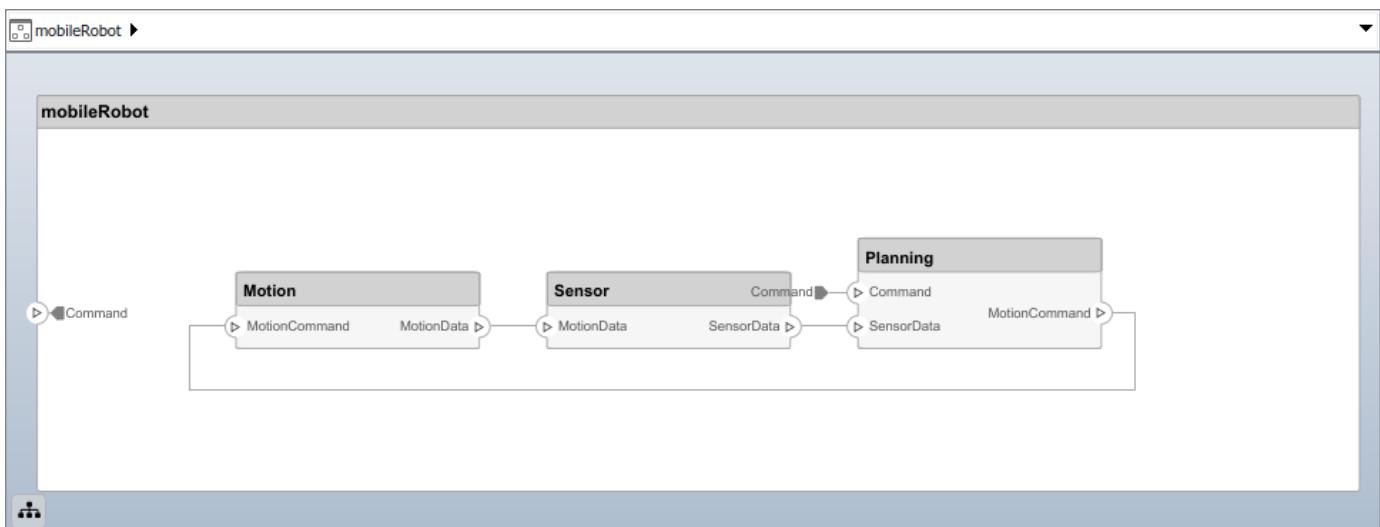
```
model.save
```

Open the model.

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

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. 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 ');
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 control');
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 interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn')

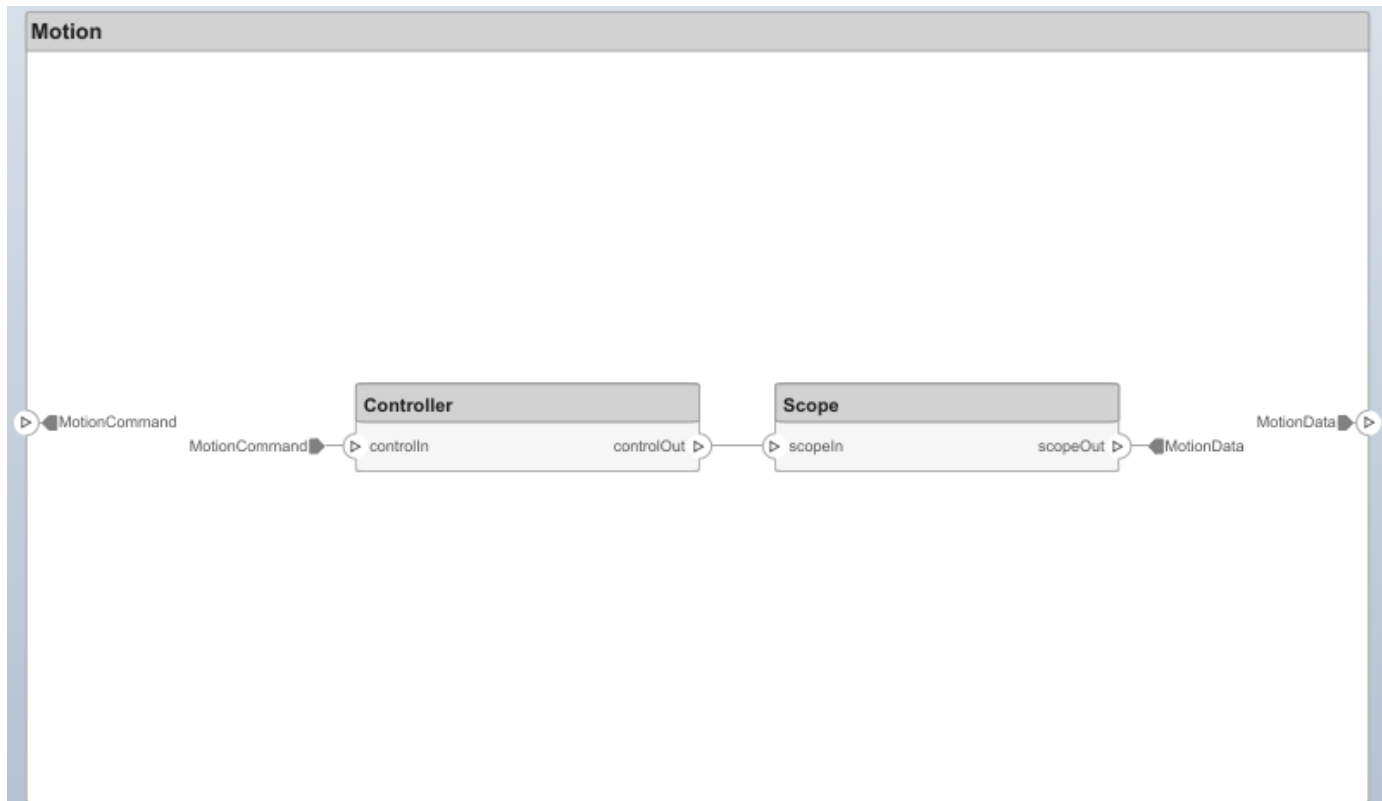
```

Save the model.

```
model.save
```

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

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



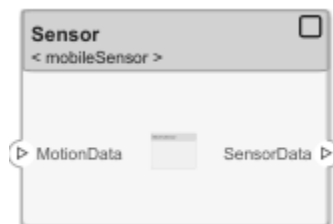
Create Model Reference

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 **Sensor** component into a reference component to reference the new model. To add additional ports on the **Sensor** component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

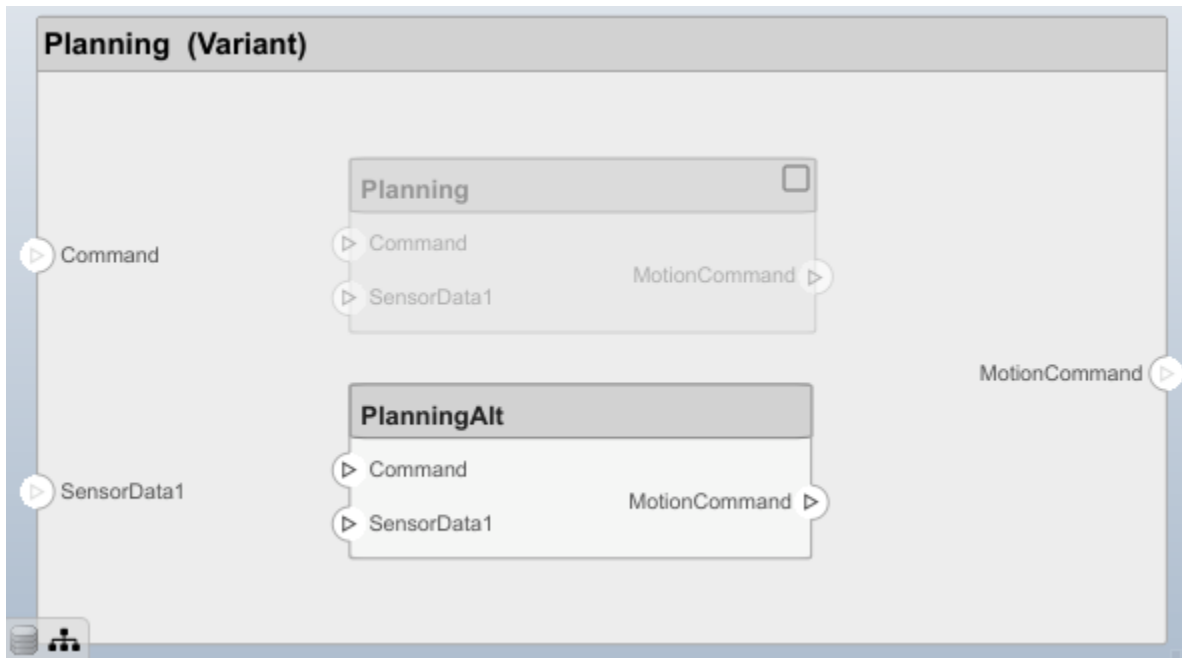
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`addElement` | `removeElement` | `getElement` | `systemcomposer.ValueType` | `systemcomposer.interface.Dictionary` | `systemcomposer.interface.DataInterface`

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021b

systemcomposer.interface.DataInterface

Data interface

Description

A DataInterface object represents the structure of a data interface.

Creation

Create a data interface.

```
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.

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

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 interface and through all operations that preserve the UUID.

Data Types: char

Object Functions

addElement	Add data element
getElement	Get object for data element
removeElement	Remove data element
setName	Set name for value type, interface, or element
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 component
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, and value types, then add components, ports, and connections. 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. 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 S
```

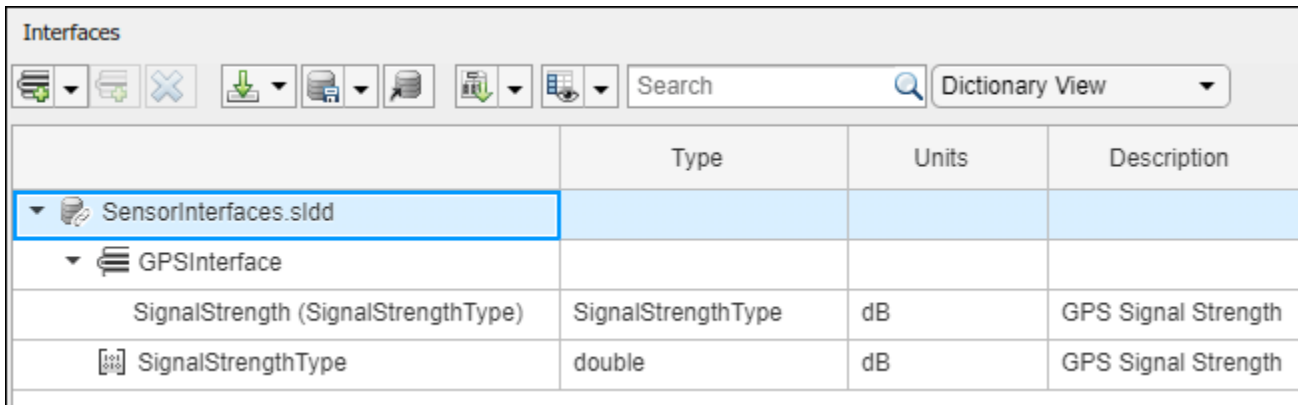


```
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.sidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



The screenshot shows the 'Interfaces' editor window. At the top, there is a toolbar with various icons and a search field. Below the toolbar is a table with the following columns: Type, Units, and Description. The table content is as follows:

	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```



```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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"/> 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','interfaces');
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.

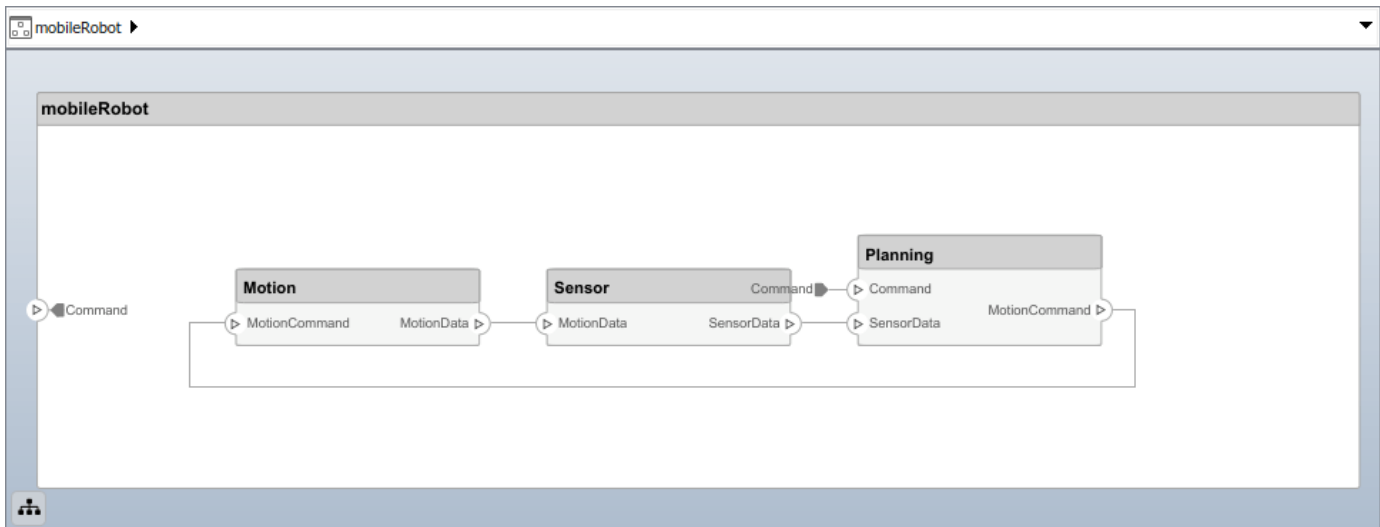
```
model.save
```

Open the model.

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

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. 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 s");
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 contr");
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 interface 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');
```



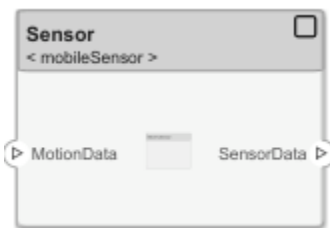
Create Model Reference

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 Sensor component into a reference component to reference the new model. To add additional ports on the Sensor component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "ElectricSensor");
linkDictionary(referenceModel, "SensorInterfaces.sldd");
referenceModel.save
```

```
linkToModel(componentSensor, "mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

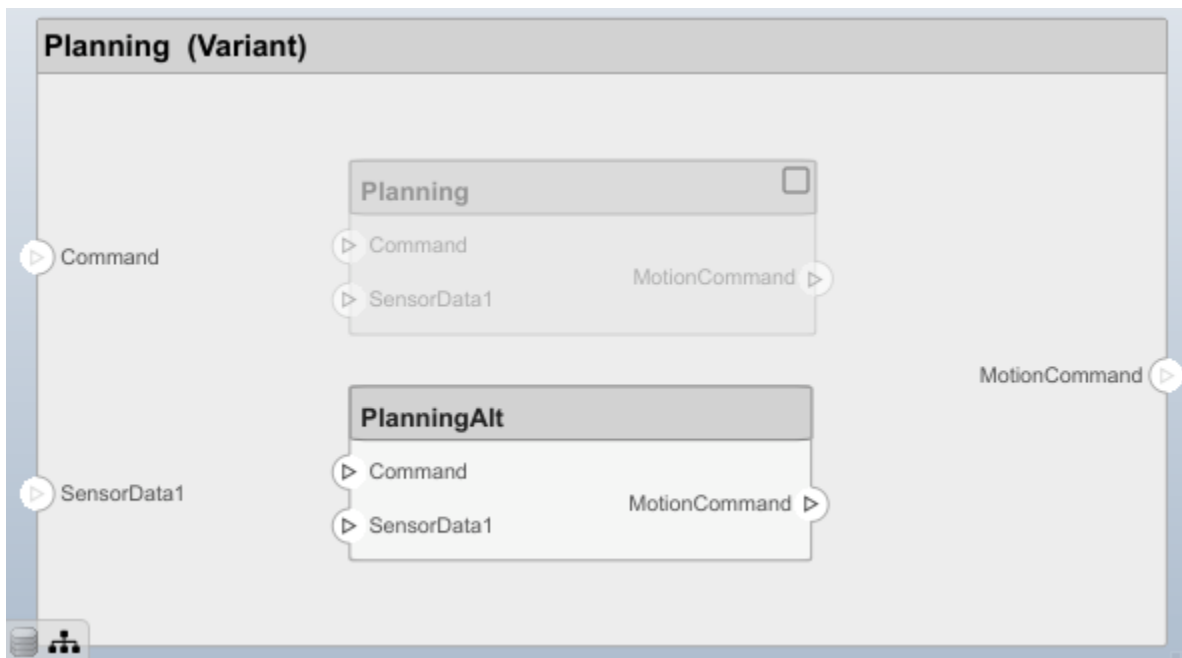
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')
% bdclose('mobileSensor')
% Simulink.data.dictionary.closeAll
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`createInterface` | `setInterface` | `addInterface` | `getInterface` | `getInterfaceNames` | `removeInterface` | `systemcomposer.ValueType` | `systemcomposer.interface.Dictionary` | `systemcomposer.interface.DataElement`

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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

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 domain and through all operations that preserve the UUID.

Data Types: char

Object Functions

destroy Remove model element

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.	“Describe 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 a 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 Simulink.ConnectionBus object that specifies at least one Simulink.ConnectionElement object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	“Specify Physical Interfaces on the 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

createInterface | addPhysicalInterface |
systemcomposer.interface.PhysicalInterface |
systemcomposer.interface.PhysicalElement | systemcomposer.interface.Dictionary

Topics

“Specify Physical Interfaces on the Ports”
“Create Interfaces”
“Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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.

```
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, interface, or element

destroy Remove model element

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.	"Describe 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 a 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 Simulink.ConnectionBus object that specifies at least one Simulink.ConnectionElement object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

[addPhysicalInterface](#) | [addElement](#) | [removeElement](#) | [getElement](#) | [systemcomposer.interface.Dictionary](#) | [systemcomposer.interface.PhysicalDomain](#) | [systemcomposer.interface.PhysicalInterface](#)

Topics

[“Specify Physical Interfaces on the Ports”](#)
[“Create Interfaces”](#)
[“Manage Interfaces with Data Dictionaries”](#)

Introduced in R2021b

systemcomposer.interface.PhysicalInterface

Physical interface

Description

A `PhysicalInterface` object represents the structure of a physical interface.

Creation

Create a physical interface.

```
interface = addPhysicalInterface(dictionary, '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.

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

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 interface and through all operations that preserve the UUID.

Data Types: char

Object Functions

addElement	Add data element
getElement	Get object for data element
removeElement	Remove data element
setName	Set name for value type, interface, or element
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 component
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

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.	“Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • "Manage Interfaces with Data Dictionaries" • "Reference Data Dictionaries"

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 Interface Editor to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.	"Create an Architecture Model with Interfaces and Requirement Links"
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • "Create Interfaces" • "Assign Interfaces to Ports"
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.	"Create Value Types as Interfaces"
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`addPhysicalInterface` | `setInterface` | `getInterface` | `getInterfaceNames` |
`removeInterface` | `systemcomposer.interface.Dictionary` |
`systemcomposer.interface.PhysicalElement` |
`systemcomposer.interface.PhysicalDomain`

Topics

“Specify Physical Interfaces on the Ports”
 “Create Interfaces”
 “Manage Interfaces with Data Dictionaries”

Introduced in R2021b

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 data element
<code>getElement</code>	Get object for data element
<code>removeElement</code>	Remove data 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 component
<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

Compatibility Considerations

`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”

Introduced in R2019a

systemcomposer.io.ModelBuilder

Model builder for System Composer architecture models

Description

Build System Composer models using a `ModelBuilder` object. Build System Composer models with components and their position in 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) % Creates the ModelBuilder object
```

Properties

Components – Component information

table

Component information, specified as a table. The table contains the hierarchical information of components, type of component (for example, Component, Reference Component, Variant Component, or Adapter), stereotypes applied on a component, and 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, their mappings to components and interfaces, as well as 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 also 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

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. Below is the mapping of information 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 the below 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
    % Name of the model to build.
    modelName = 'scExampleModelBuider';
    % Name of the profile.
    profile = 'UAVComponent';
    % 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`, 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);

% 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 case of interfaces adding the interface name to model builder.
        interfaceName = definitionContents.Name{rowItr};
        % Get 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);
        % Builder utility function to add interface to data
        % dictionary.
        builder.addInterface(interfaceName,interfaceID);
    else
        % In case of element read element properties and add the element to
        % parent interface.
        elementName = definitionContents.Name{rowItr};
        interfaceID = idService.getID('interfaces',parentInterface);
        % ElementID is unique within a interface.
        % Appending 'E' at start of ID for uniformity. The generated ID for
        % input element is unique within parent interface name as container.
        elemID = idService.getID(parentInterface,elementName,'E');
        % Datatype, dimensions, units, minimum and maximum properties of
        % 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};
    % Builder function to add element with properties in interface.
    builder.addElementInInterface(elementName,elemID,interfaceID,datatype,dimensions,units,
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 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};
    % Populating the contents of table using the builder.
    if strcmp(class,'component')
        ID = idService.getID('comp',Name);
        % Root ID is by default set as zero.
        if strcmp(Parent,'scExampleSmallUAV')
            parentID = "0";
        else
            parentID = idService.getID('comp',Parent);
        end
        % Builder utility function to add component.
        builder.addComponent(Name,ID,parentID);
        % Reading the property values
        kind = excelContents.Kind{rowItr};
        domain = excelContents.Domain{rowItr};
        % *Builder to set stereotype and property values.
        builder.setComponentProperty(ID,'StereotypeName','UAVComponent.PartDescriptor','ModelName
else
    % In this example, concatenation of 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. compID is assumed as "0".
    if strcmp(Parent,'scExampleSmallUAV')
        compID = "0";
    else
        compID = idService.getID('comp',Parent);
    end
    % Builder utility function to add port.
    builder.addPort(Name,class,portID,compID );

    % InterfaceName specifies the name of the interface linked to port.
    interfaceName = excelContents.InterfaceName{rowItr};

    % Get interface ID. getID() will return the same IDs already
    % generated while adding interface in Step 2.
    interfaceID = idService.getID('interfaces',interfaceName);
    % Builder to map interface to port.
    builder.addInterfaceToPort(interfaceID,portID);

    % Reading the connectedTo information to build connections between

```

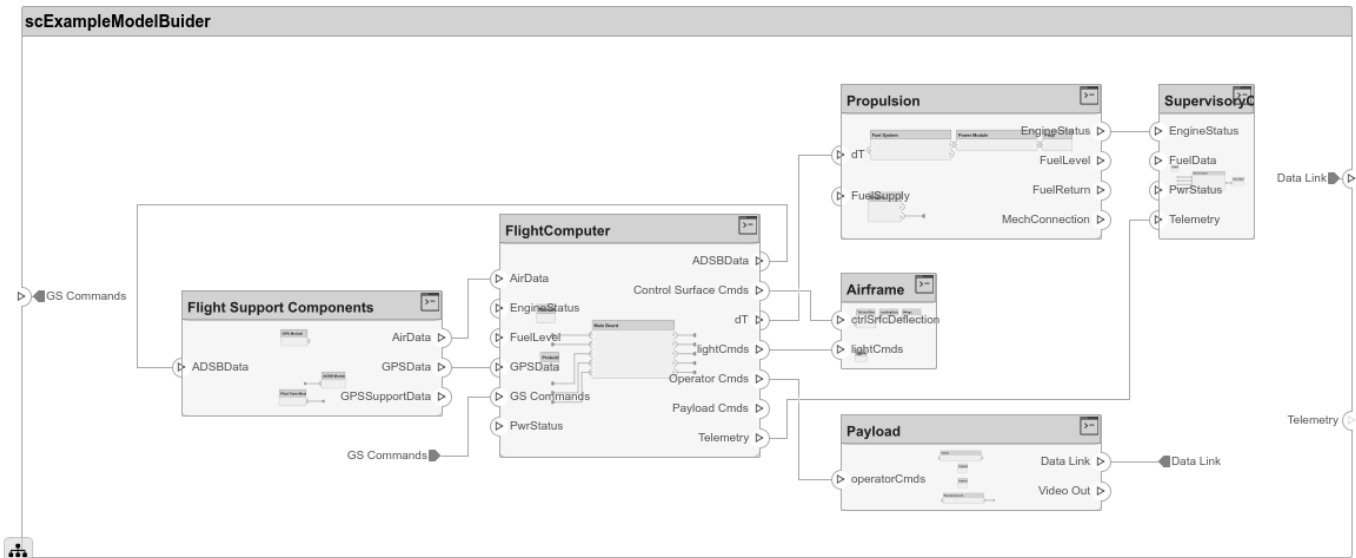
```

% components.
connectedTo = excelContents.ConnectedTo{rowItr};
% connectedTo is in format:
% (DestinationComponentName::DestinationPortName).
% For this example, considering 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
    % port name and parent component name. If port id is already
    % generated getID() function returns the same id for input key.
    connectedPortID = idService.getID('port',strcat(splits(2),splits(1)));
    % Using builder to populate connection table.
    sourcePortID = portID;
    destPortID = connectedPortID;
    % Builder 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);
```



Close Model

```
bdclose(modelName)
```

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> <i>Component ports</i> are interaction points on the component to other components. <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	“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 anonymous interface cannot be another interface name. Anonymous 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.

See Also

`importModel` | `exportModel`

Topics

“Import and Export Architecture Models”

Introduced in R2019b

systemcomposer.profile.Profile

Profile

Description

A Profile object represents a profile for a System Composer model.

Creation

Create a profile.

```
profile = systemcomposer.profile.Profile.createProfile('profileName');
```

Properties

Name — Name of profile

character vector

Name of profile, specified as a character vector. This property must be a valid MATLAB identifier.

Data Types: char

FriendlyName — Descriptive name of profile

character vector

Descriptive name of profile, specified as a character vector. This property can contain spaces and special characters, but no new lines.

Data Types: char

Description — Description text for profile

multi-line character vector

Description text for profile, specified as a multi-line character vector.

Data Types: char

Stereotypes — Stereotypes

array of stereotype objects

Stereotypes defined in profile, specified as an array of `systemcomposer.profile.Stereotype` objects.

Data Types: char

Object Functions

<code>createProfile</code>	Create profile
<code>addStereotype</code>	Add stereotype to profile
<code>removeStereotype</code>	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, and value types, then add components, ports, and connections. 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;
```

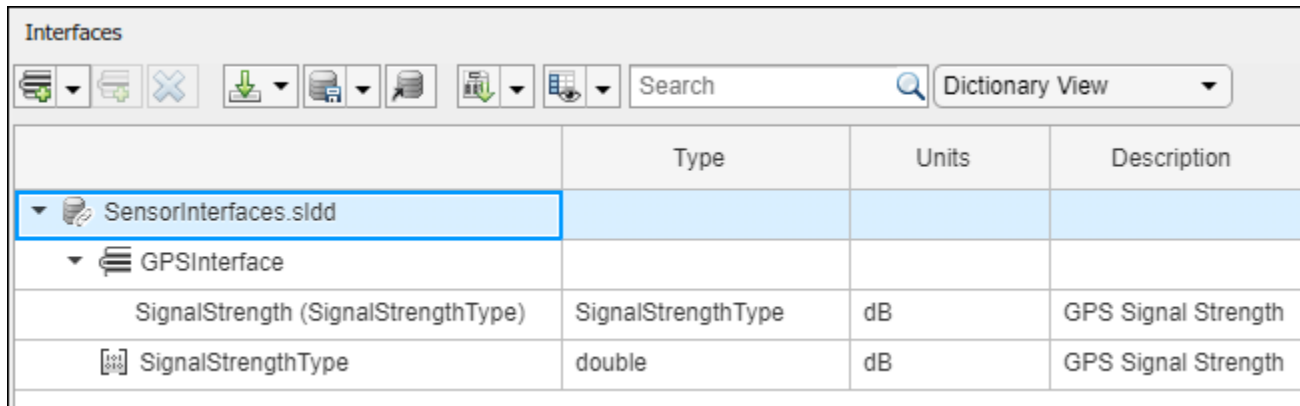
Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Link the data dictionary to the model.

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

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

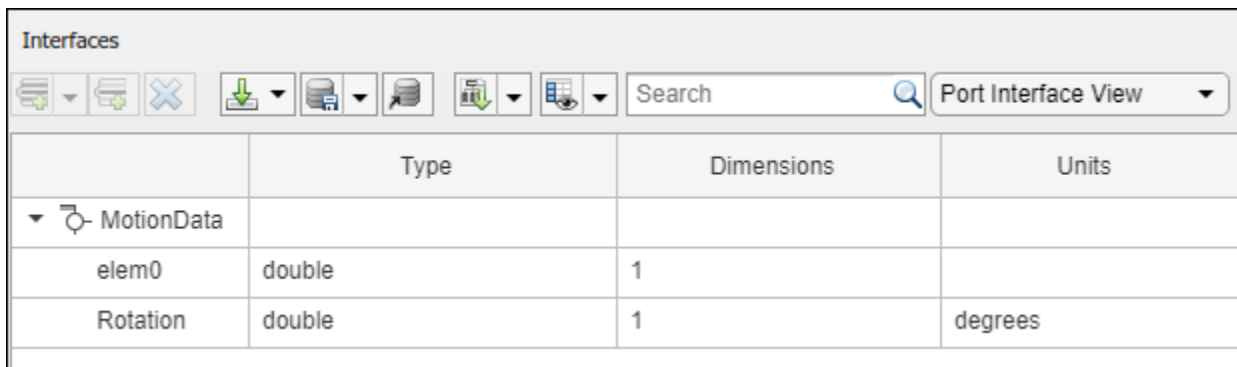
```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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','interfaces');
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.

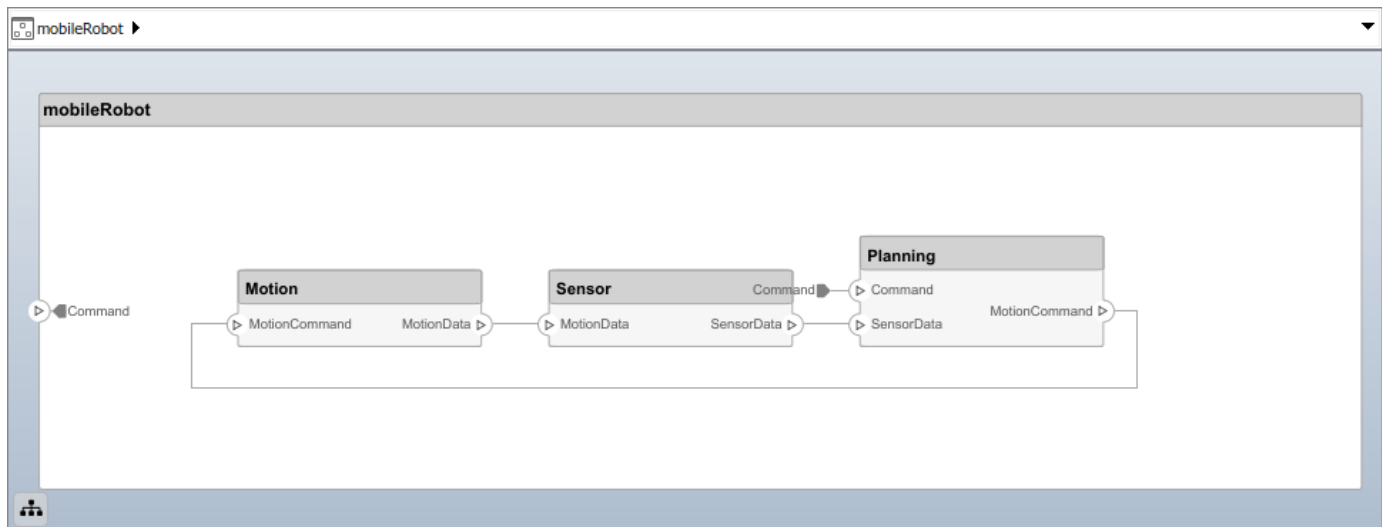
```
model.save
```

Open the model.

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

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. 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');
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 control');
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 interface 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');
```



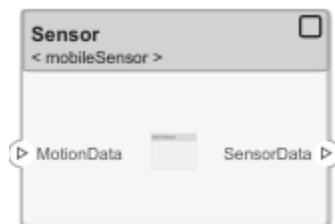
Create Model Reference

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 **Sensor** component into a reference component to reference the new model. To add additional ports on the **Sensor** component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

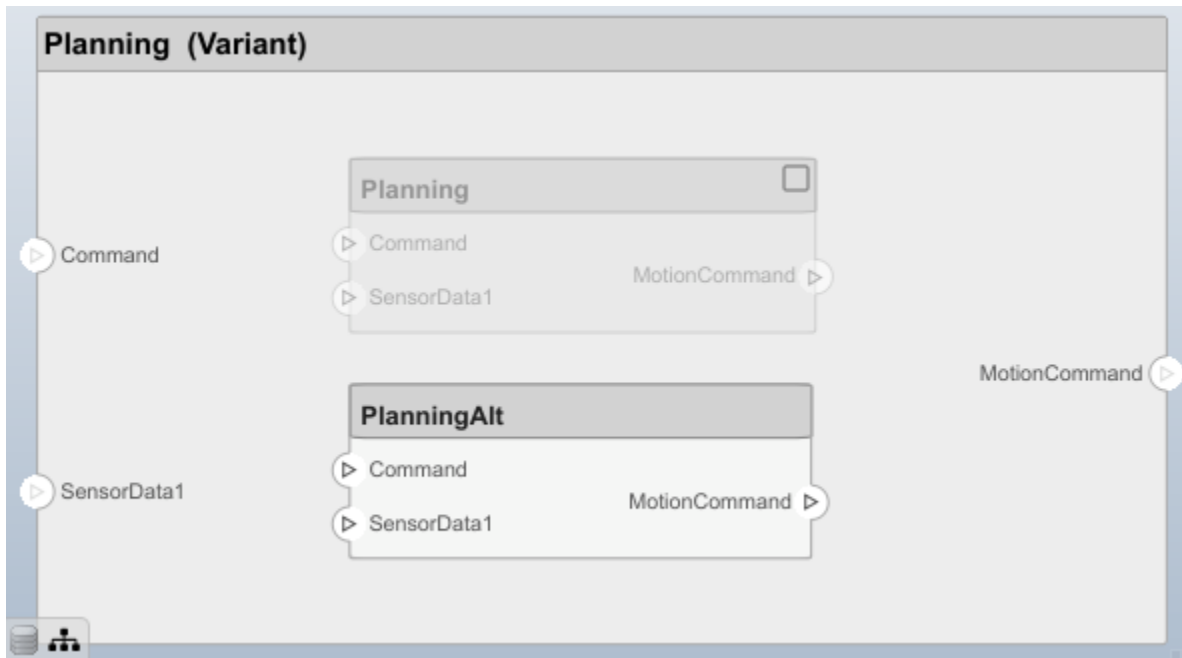
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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

Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

editor | systemcomposer.profile.Stereotype | systemcomposer.profile.Property | loadProfile

Topics

“Define Profiles and Stereotypes”
 “Use Stereotypes and Profiles”

Introduced in R2019a

systemcomposer.profile.Property

Property in a 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.

```
addProperty(stereotype,AttributeName,'DefaultValue',AttributeValue)
```

Properties

Name — Name of property

character vector

Name of property, specified as a character vector. This property must be a valid MATLAB identifier.

Data Types: char

Type — Property data type

character vector

Property data type, specified as a character vector with a valid data type.

Data Types: char

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

Property units, specified as a character vector.

Data Types: `char`

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

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, and value types, then add components, ports, and connections. 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. 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);
linkDictionary(model, "SensorInterfaces.sidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)



componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)

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"/> 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','interfaces');
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.

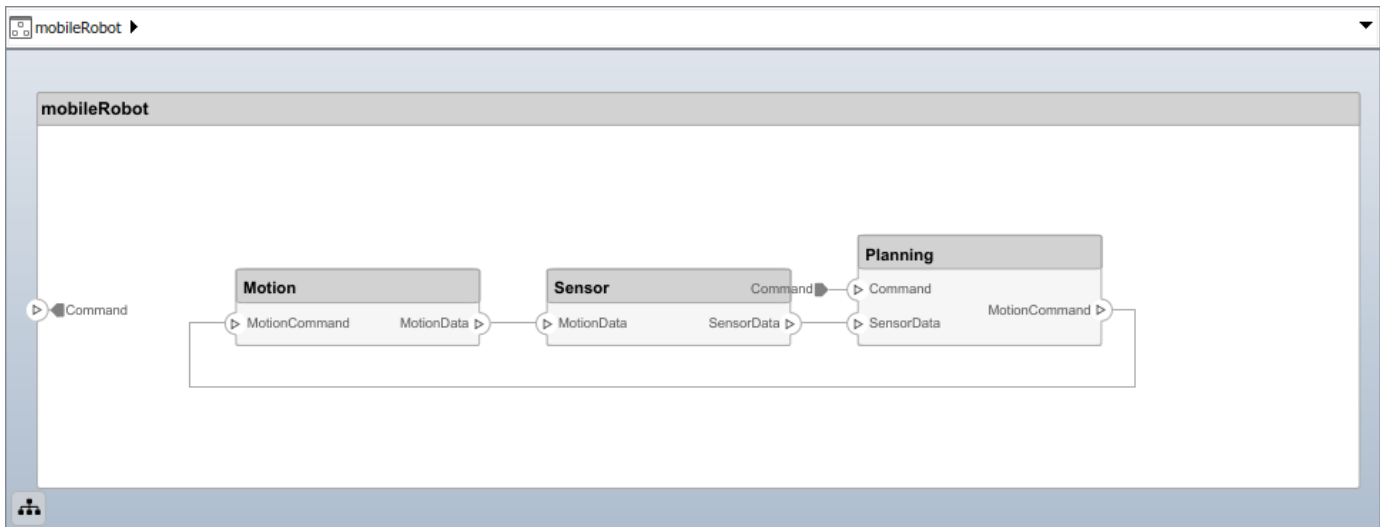
```
model.save
```

Open the model.

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

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. 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 s");
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 contr");
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 interface 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');
```



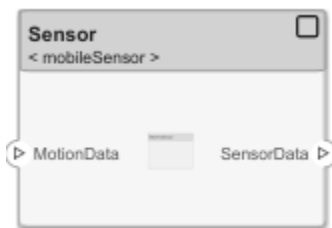
Create Model Reference

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 Sensor component into a reference component to reference the new model. To add additional ports on the Sensor component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "ElectricSensor");
linkDictionary(referenceModel, "SensorInterfaces.sldd");
referenceModel.save
```

```
linkToModel(componentSensor, "mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

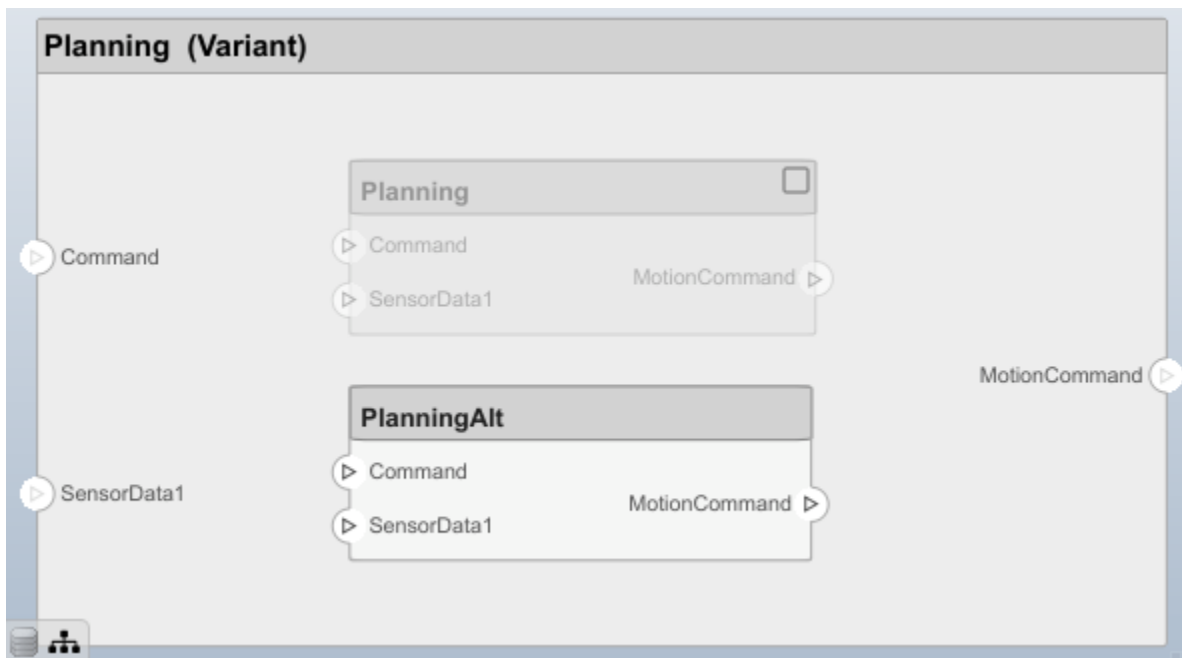
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')
% bdclose('mobileSensor')
% Simulink.data.dictionary.closeAll
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

systemcomposer.profile.Stereotype | systemcomposer.profile.Profile | removeProperty | addProperty

Topics

“Define Profiles and Stereotypes”
 “Use Stereotypes and Profiles”

Introduced in R2019a

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.

```
addStereotype(profile, 'name')
```

Properties

Name — Name of stereotype

character vector

Name of stereotype, specified as a character vector. This property must be a valid MATLAB identifier.

Example: 'HardwareComponent'

Data Types: char

Description — Description text for stereotype

character vector

Description text for stereotype, specified as a character vector.

Data Types: char

Icon — Icon name for stereotype

character vector

Icon name for stereotype, specified as a character vector.

Example: 'default'

Example: 'application'

Example: 'channel'

Example: 'controller'

Example: 'database'

Example: 'devicedriver'

Example: 'memory'

Example: 'network'

Example: 'plant'

Example: 'sensor'

Example: 'subsystem'

Example: 'transmitter'

Data Types: char

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

'Component' | 'Port' | 'Connector' | 'Interface'

Element type to which stereotype can be applied, specified as a character vector of the following options: 'Component', 'Port', 'Connector', or 'Interface'.

Data Types: char

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].

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].

Example: [206 232 246]

Data Types: uint32

ConnectorLineStyle — Connector line style

character vector

Connector line style name, specified as a character vector.

Example: 'Default'

Example: 'Dot'

Example: 'Dash'

Example: 'Dash Dot'

Example: 'Dash Dot Dot'

Data Types: char

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 property objects

Owned properties contained in stereotype, specified as a cell array of character vectors or an array of `systemcomposer.profile.Property` objects. The owned properties do not include properties inherited from the stereotype base hierarchy.

Data Types: char

Object Functions

<code>addProperty</code>	Define custom property for stereotype
<code>removeProperty</code>	Remove property from stereotype
<code>getDefaultElementStereotype</code>	Get default stereotype for elements
<code>setDefaultElementStereotype</code>	Set default stereotype for elements
<code>find</code>	Find stereotype by 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, and value types, then add components, ports, and connections. 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;
```

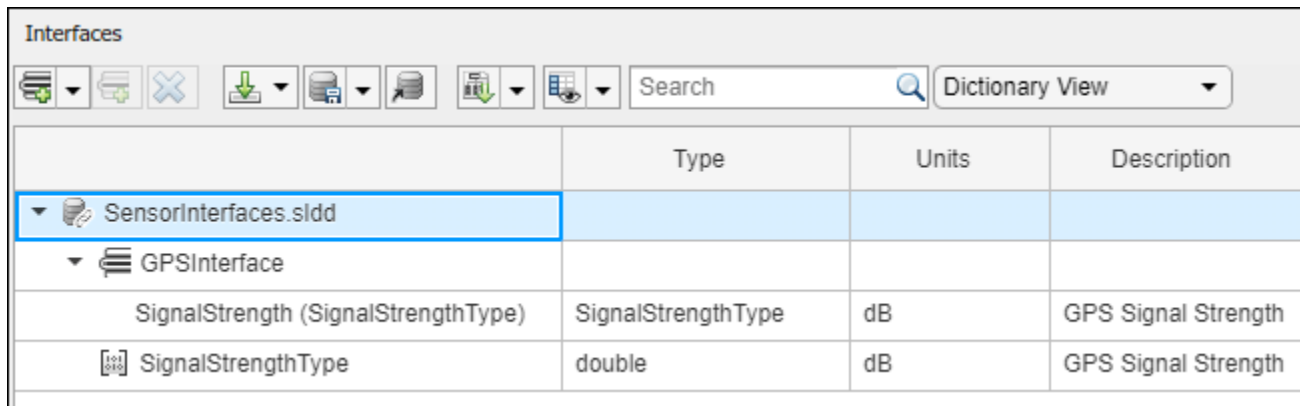
Create an interface data dictionary and add a data interface. Add a data element to the data interface. Add a value type to the interface data dictionary. Assign the type of the data element to the value type. Link the data dictionary to the model.

```
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);
linkDictionary(model, "SensorInterfaces.sidd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.



The screenshot shows the 'Interfaces' editor window. At the top, there are several icons for file operations and a search bar. Below the search bar is a dropdown menu set to 'Dictionary View'. The main area contains a table with the following structure:

	Type	Units	Description
▼ SensorInterfaces.sidd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

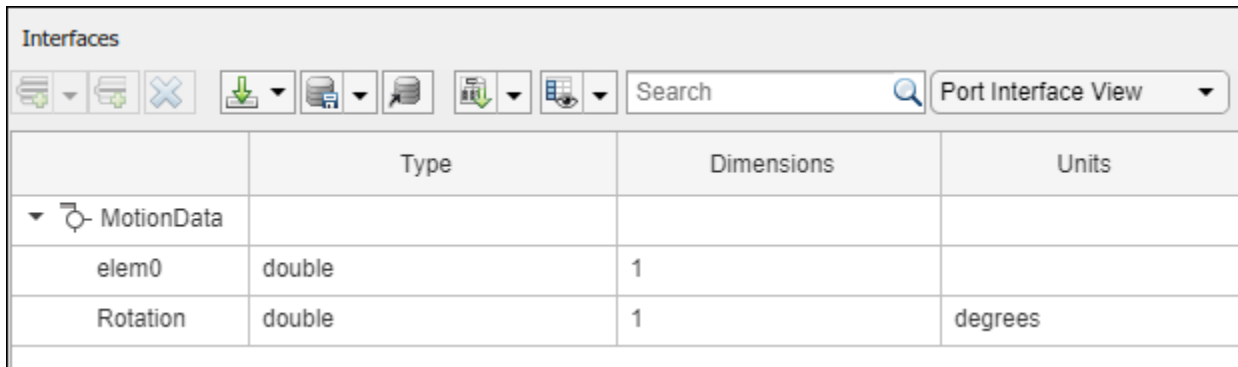
```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```



```
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"/> 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',"interfaces");
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.

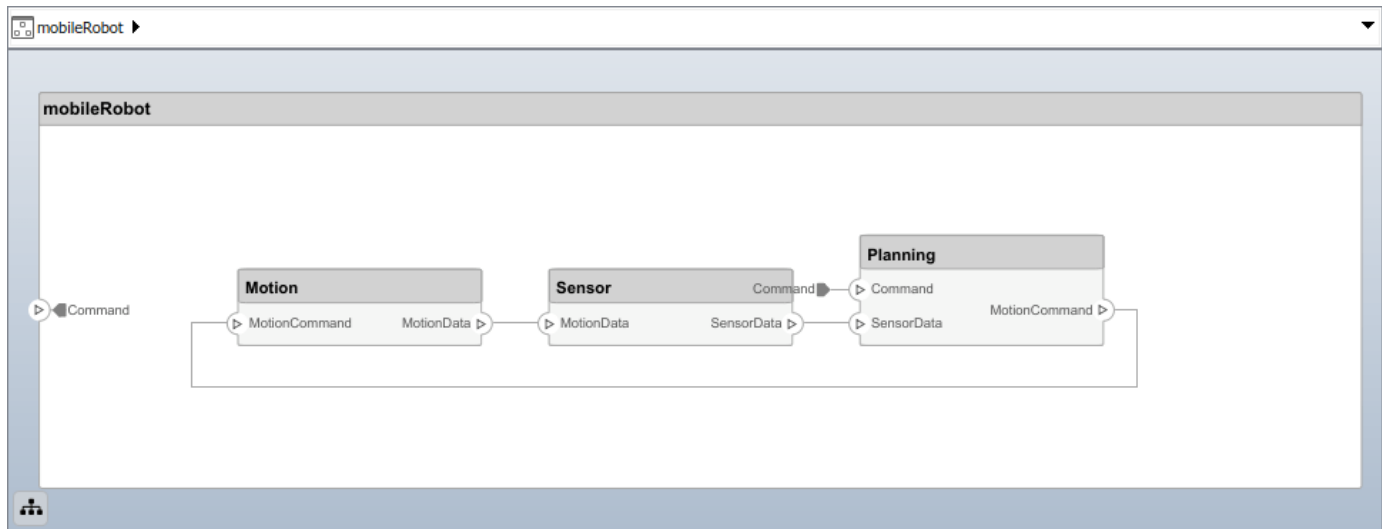
```
model.save
```

Open the model.

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

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. 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 ");
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 control ");
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 interface 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');
```



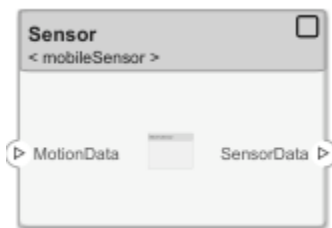
Create Model Reference

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 Sensor component into a reference component to reference the new model. To add additional ports on the Sensor component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch, "ElectricSensor");
linkDictionary(referenceModel, "SensorInterfaces.sldd");
referenceModel.save
```

```
linkToModel(componentSensor, "mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

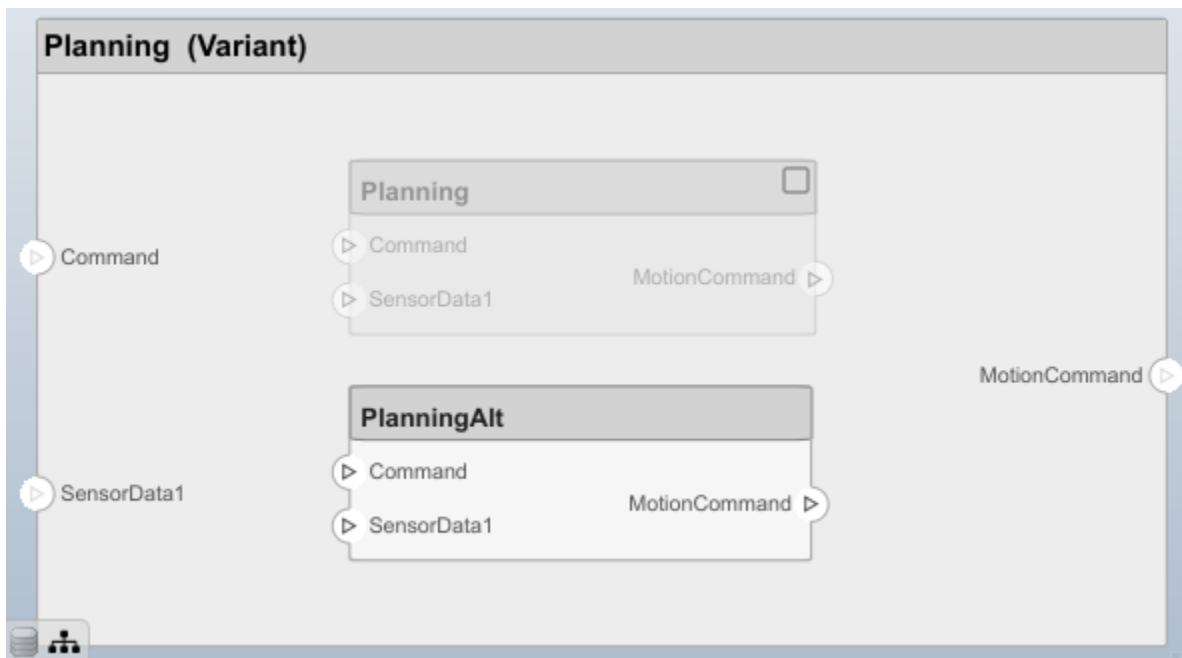
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')
% bdclose('mobileSensor')
% Simulink.data.dictionary.closeAll
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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 elements: root-level architecture, component architecture, connectors, ports, data interfaces, and value types of a model. 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. For each element the stereotype is applied to, specific property values are specified.	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.	<ul style="list-style-type: none"> • “Set Properties” • “Add Properties with Stereotypes” • “Set Properties for Analysis”
profile	A profile is a package of stereotypes 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 profile or in several. Profiles are stored in XML files when they are saved.	<ul style="list-style-type: none"> • “Define Profiles and Stereotypes” • “Use Stereotypes and Profiles”

See Also

[addStereotype](#) | [getStereotype](#) | [removeStereotype](#) | [systemcomposer.profile.Profile](#)

Topics

“Define Profiles and Stereotypes”

“Use Stereotypes and Profiles”

Introduced in R2019a

systemcomposer.query.Constraint

Query constraint

Description

The `Constraint` class is the base class for all System Composer query constraints.

Object Functions

<code>AnyComponent</code>	Create query to select all components in model
<code>IsStereotypeDerivedFrom</code>	Create query to select stereotype derived from qualified name
<code>HasStereotype</code>	Create query to select architecture elements with stereotype based on specified sub-constraint
<code>HasPort</code>	Create query to select architecture elements with port on component based on specified sub-constraint
<code>HasInterface</code>	Create query to select architecture elements with interface on port based on specified sub-constraint
<code>HasInterfaceElement</code>	Create query to select architecture elements with interface element on interface based on specified sub-constraint
<code>IsInRange</code>	Create query to select range of property values
<code>Property</code>	Create query to select non-evaluated values for object properties or stereotype properties for elements
<code>PropertyValue</code>	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.

Open the model.

```
import systemcomposer.query.*

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/FOB Locator System/FOB Locator Module'      }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
    {'KeylessEntryArchitecture/Sound System/Sound Controller'              }
    {'KeylessEntryArchitecture/Lighting System/Lighting Controller'         }
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller'}
```



```
compObjs=1x5 object
```

```
1x5 Component array with properties:
```

```
IsAdapterComponent
Architecture
ReferenceName
Name
Parent
Ports
OwnedPorts
OwnedArchitecture
Position
Model
SimulinkHandle
SimulinkModelHandle
UUID
ExternalUID
```

```
% Include reference models in the search
```

```
softwareComps = model.find(con1, 'IncludeReferenceModels', true)
```

```
softwareComps = 9x1 cell
```

```
{'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller'}
{'KeylessEntryArchitecture/Sound System/Sound Controller'}
{'KeylessEntryArchitecture/Lighting System/Lighting Controller'}
{'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Sensor/Detect Door L...'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Driver Door Lock Sensor/Detect Door L...'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Sensor/Detect Door L...'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Sensor/Detect Door L...'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Sensor/Detect Door L...'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Sensor/Detect Door L...'}
{'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 Driver Door Lock Actuator'}
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Actuator'}
```

Find all the base components in the system.

```
con2 = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.BaseComponent"));
```

```
baseComps = model.find(con2)
```

```
baseComps = 18x1 cell
```

```
{'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module'} }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller'} }
{'KeylessEntryArchitecture/Sound System/Sound Controller'} }
{'KeylessEntryArchitecture/Lighting System/Lighting Controller'} }
{'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller'} }
{'KeylessEntryArchitecture/Engine Control System/Start//Stop Button'} }
{'KeylessEntryArchitecture/Sound System/Dashboard Speaker'} }
{'KeylessEntryArchitecture/FOB Locator System/Center Receiver'} }
{'KeylessEntryArchitecture/FOB Locator System/Front Receiver'} }
{'KeylessEntryArchitecture/FOB Locator System/Rear 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/Rear Driver Door Lock Sensor'} }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Sensor'} }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Actuator'} }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Actuator'} }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Actuator'} }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Actuator'}
```

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/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' }
```

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/FOB Locator System/FOB Locator Module' }
    {'KeylessEntryArchitecture/Sound System/Sound Controller' }
    {'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/Door Lock//Unlock System/Door Lock Controller' }
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start 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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • "Create Architecture Views Interactively" • "Create Architectural Views Programmatically"
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"> • <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. • <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. 	“Display Component Hierarchy and Architecture Hierarchy Using Views”

See Also

find | createView | modifyQuery | runQuery | removeQuery

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

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.

```
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 of 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, 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
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 component
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, and value types, then add components, ports, and connections. 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. 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 S');
element.setType(valueType);
linkDictionary(model, "SensorInterfaces.sldd");
```

Save the changes to the interface data dictionary.

```
dictionary.save
```

View the interfaces in the Interface Editor.

	Type	Units	Description
▼ SensorInterfaces.sldd			
▼ GPSInterface			
SignalStrength (SignalStrengthType)	SignalStrengthType	dB	GPS Signal Strength
SignalStrengthType	double	dB	GPS Signal Strength

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

```
componentSensor = addComponent(arch, 'Sensor');
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
```

```
componentPlanning = addComponent(arch, 'Planning');
planningPorts = addPort(componentPlanning.Architecture, {'Command', 'SensorData1', 'MotionCommand'});
planningPorts(2).setInterface(interface)
```

```
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','interfaces');
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.

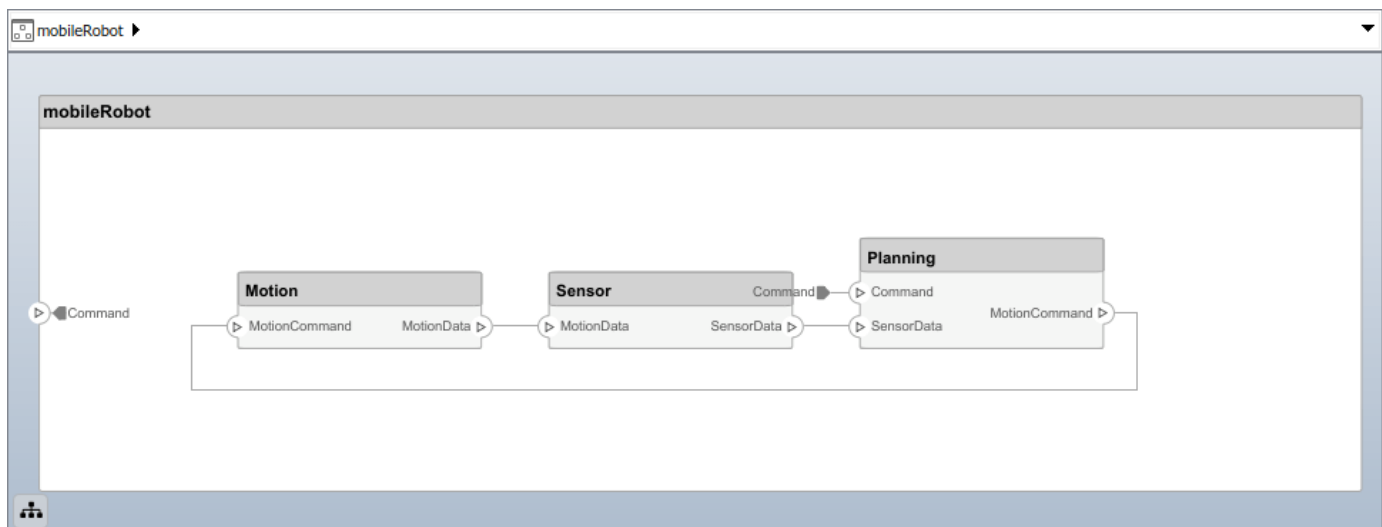
```
model.save
```

Open the model.

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

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. 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');
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 control');
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 interface element must be specified
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn')

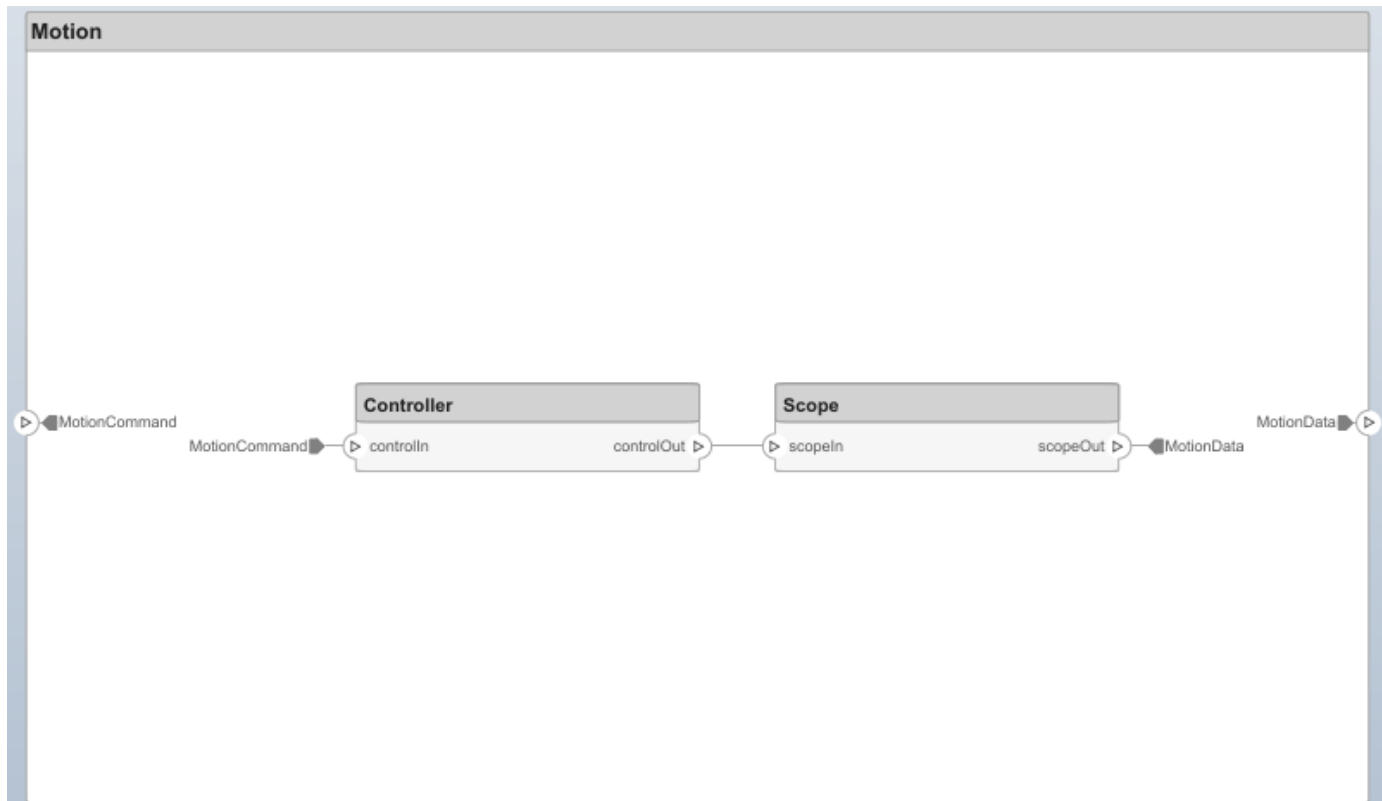
```

Save the model.

```
model.save
```

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

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



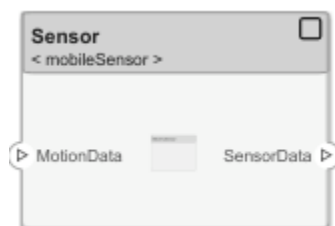
Create Model Reference

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 **Sensor** component into a reference component to reference the new model. To add additional ports on the **Sensor** component, you must update the referenced model "mobileSensor".

```
referenceModel = systemcomposer.createModel("mobileSensor");
referenceArch = referenceModel.Architecture;
newComponents = addComponent(referenceArch,"ElectricSensor");
linkDictionary(referenceModel,"SensorInterfaces.slidd");
referenceModel.save
```

```
linkToModel(componentSensor,"mobileSensor");
```



Apply a stereotype to the architecture and component of the linked reference model.

```
referenceModel.applyProfile("GeneralProfile");
referenceArch.applyStereotype("GeneralProfile.softwareComponent");
batchApplyStereotype(referenceArch, 'Component', "GeneralProfile.projectElement")
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(componentSensor.Architecture, {'MotionData', 'SensorData'}, {'in', 'out'});
sensorPorts(2).setInterface(interface)
connect(arch, componentSensor, componentPlanning, 'Rule', 'interfaces');
connect(arch, componentMotion, componentSensor);
```

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(componentPlanning);
```

Add an additional variant choice named `PlanningAlt`. 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, {'PlanningAlt'}, {'PlanningAlt'});
```

Create the necessary ports on `PlanningAlt`.

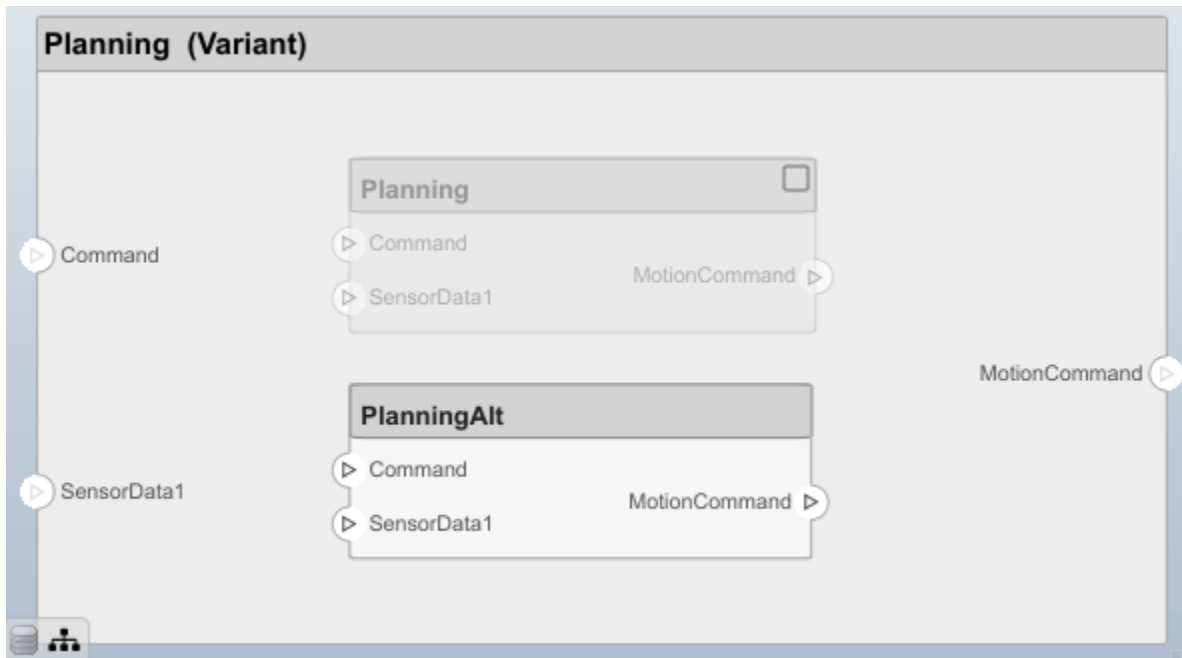
```
setActiveChoice(variantComp, choice2)
planningAltPorts = addPort(choice2.Architecture, {'Command', 'SensorData1', 'MotionCommand'}, {'in',
planningAltPorts(2).setInterface(interface)
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp, 'PlanningAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

Uncomment this code and run it to clean up the artifacts created by this example.

```
% bdclose('mobileRobotAPI')  
% bdclose('mobileSensor')  
% Simulink.data.dictionary.closeAll  
% systemcomposer.profile.Profile.closeAll
```

```
% delete('Profile.xml')
% delete('SensorInterfaces.sldd')
```

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”

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 Interface Editor so that you can reuse the value types as interfaces or data elements.	“Create Value Types as Interfaces”
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

`addValueType` | `systemcomposer.interface.DataInterface` | `systemcomposer.interface.Dictionary` | `systemcomposer.interface.DataElement`

Topics

“Create Interfaces”

“Manage Interfaces with Data Dictionaries”

Introduced in R2021b

systemcomposer.view.BaseViewComponent

(Removed) View components

Note The `systemcomposer.view.BaseViewComponent` 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

The `BaseViewComponent` class inherits from the `systemcomposer.view.ViewElement` class.

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')`

Compatibility Considerations

systemcomposer.view.BaseViewComponent class has been removed

Errors starting in R2021a

The `systemcomposer.view.BaseViewComponent` class is removed in R2021a with the introduction of new views APIs. 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”

Introduced in R2019b

systemcomposer.view.ComponentOccurrence

(Removed) Shadow of component from composition in view

Note The `systemcomposer.view.ComponentOccurrence` 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

The `ComponentOccurrence` class inherits from the `systemcomposer.view.BaseViewComponent` class.

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')`

Compatibility Considerations

systemcomposer.view.ComponentOccurrence class has been removed

Errors starting in R2021a

The `systemcomposer.view.ComponentOccurrence` class is removed in R2021a with the introduction of new views APIs. 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”

Introduced in R2019b

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 and get the `Root` property.

```
objView = createView(objModel);  
objElemGroup = objView.Root
```

The `createView` method is the constructor for the `systemcomposer.view.View` class. The `Root` property returns the `systemcomposer.view.ElementGroup` that defines the view.

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.

```
sckKeylessEntrySystem
```

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.

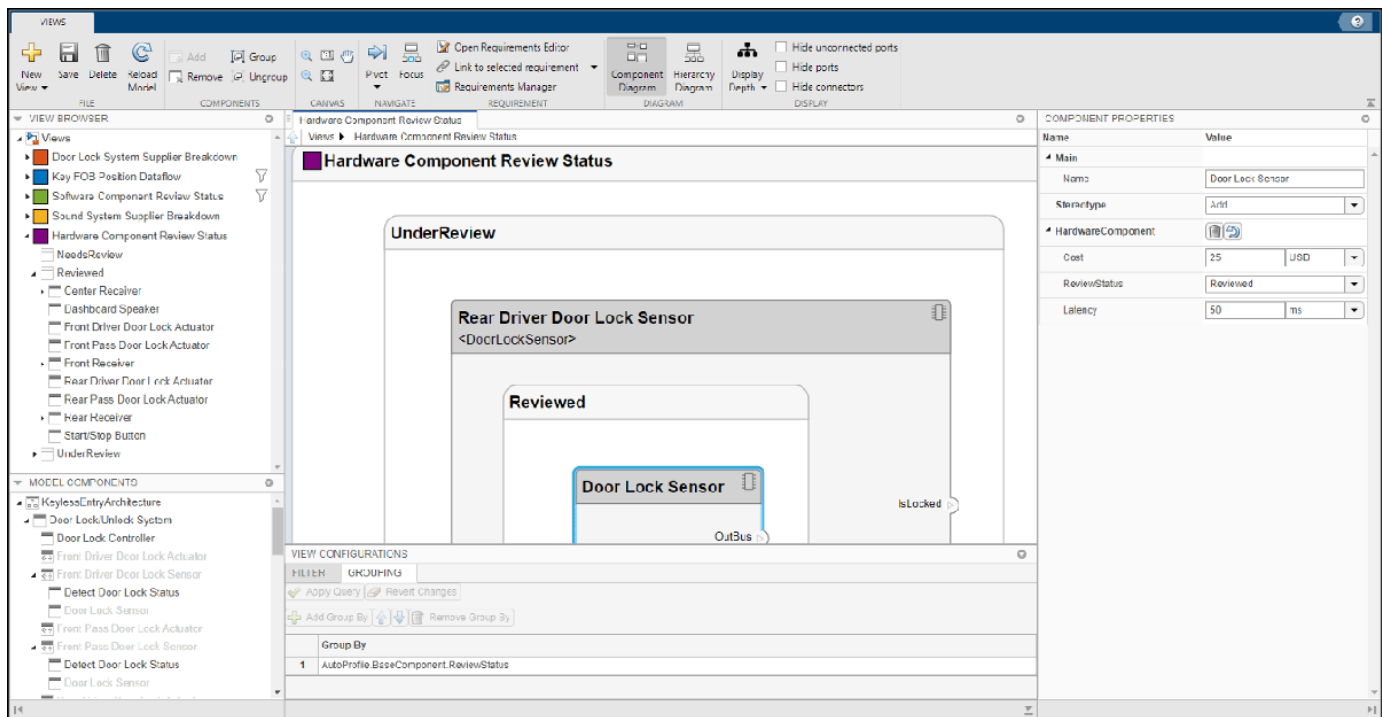
```
hwCompQuery = HasStereotype(IsStereotypeDerivedFrom('AutoProfile.HardwareComponent'));
```

2. Use the query to create a view.

```
model.createView('Hardware Component Review Status',...
  'Select',hwCompQuery,... % Query to use for the selection
  'GroupBy',{'AutoProfile.BaseComponent.ReviewStatus'},... % Stereotype property to qualify by
  'IncludeReferenceModels',true,... % Include components in referenced models
  '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'.

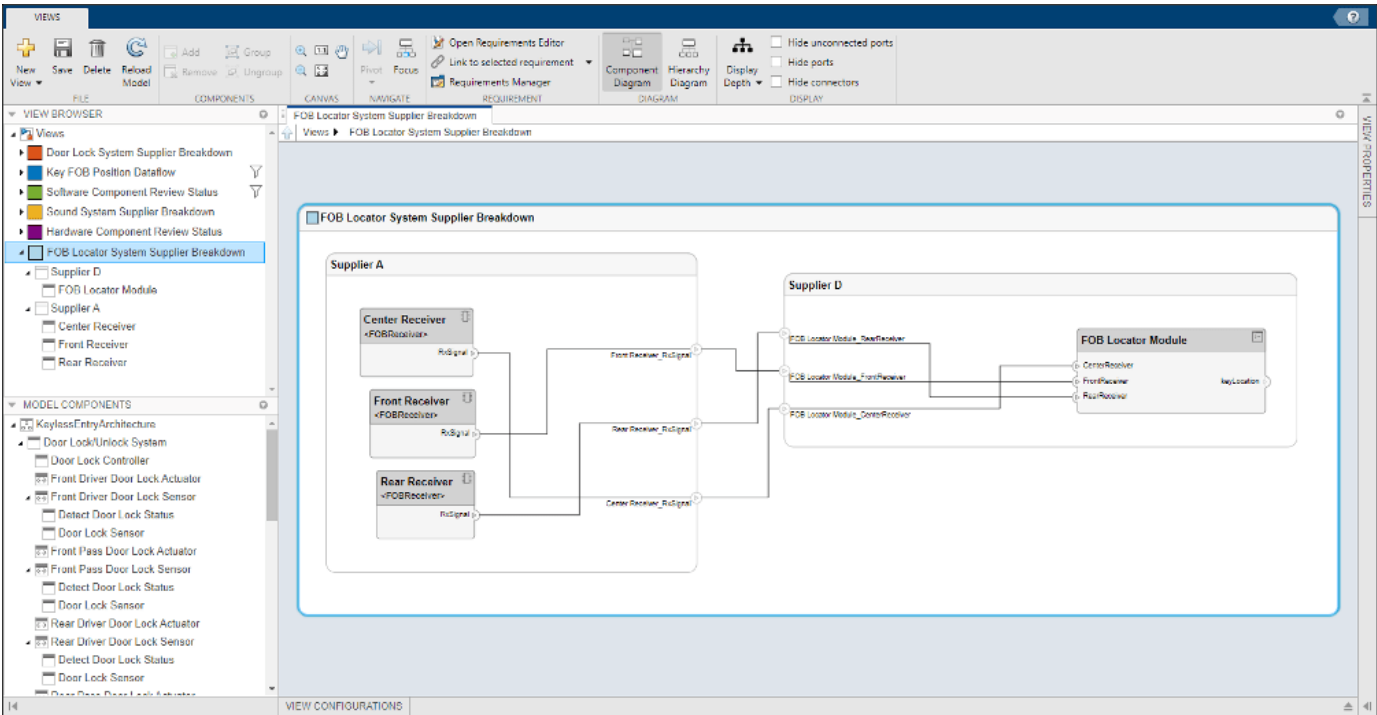
```
supplierA = fobSupplierView.Root.createSubGroup('Supplier A');
```

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	"Display Component Hierarchy and Architecture Hierarchy Using Views"

See Also

openViews | createView | getView | deleteView | systemcomposer.view.View

Topics

"Create Architecture Views Interactively"

"Create Architectural Views Programmatically"

"Display Component Hierarchy and Architecture Hierarchy Using Views"

Introduced in R2021a

systemcomposer.view.View

Architecture view

Description

A View object is used to manage architecture views for a System Composer model.

Creation

Create a view.

```
objView = createView(objModel)
```

The `createView` method is the constructor for the `systemcomposer.view.View` class.

Properties

Name — Name of view

character vector

Name of view, specified as a character vector.

Example: 'NewView'

Data Types: char

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 architecture

character vector

Color of view architecture, specified as a character vector. 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.

Example: `color = get(objViewArchitecture, 'Color')`

Description — Description of view architecture

character vector

Description of view architecture, specified as a character vector.

Example: `description = get(objView, 'Description')`

Example: `set(objView, 'Description', description)`

Data Types: char

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.

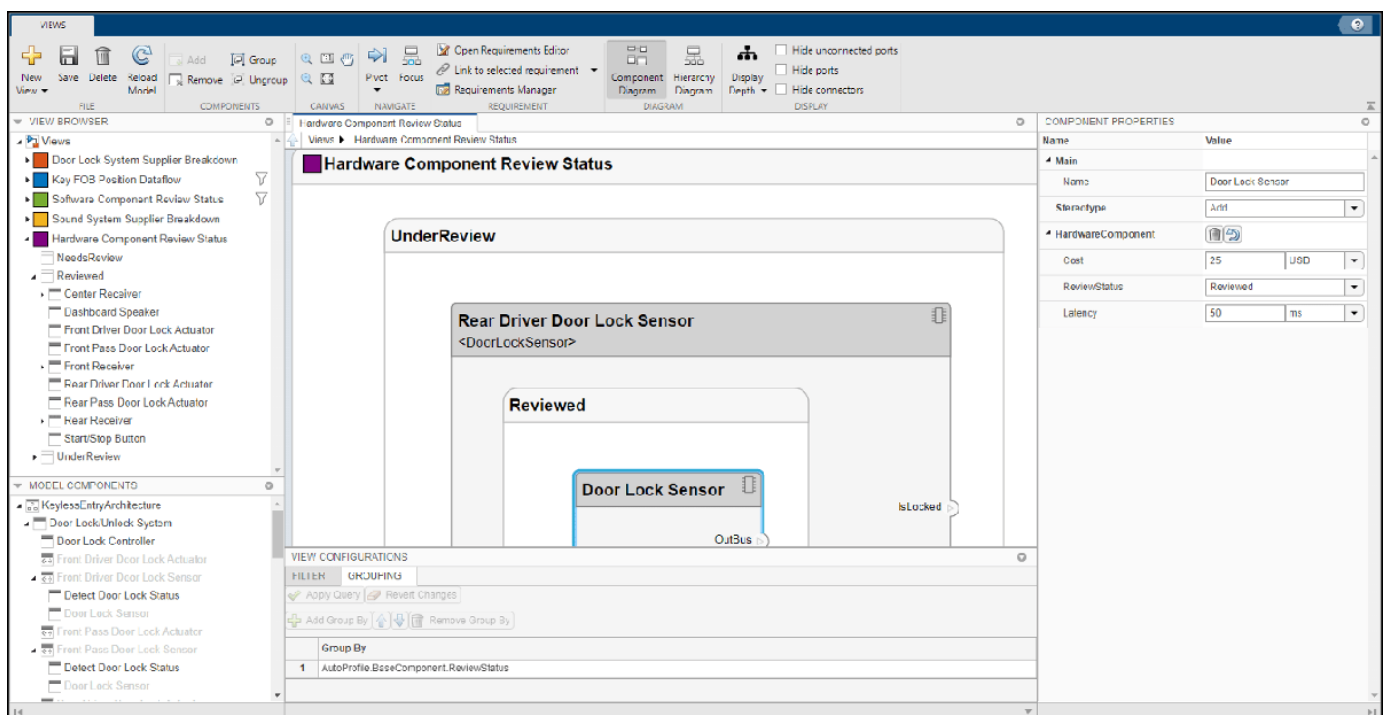
```
hwCompQuery = HasStereotype(IsStereotypeDerivedFrom('AutoProfile.HardwareComponent'));
```

2. Use the query to create a view.

```
model.createView('Hardware Component Review Status',...
  'Select',hwCompQuery,... % Query to use for the selection
  'GroupBy',{'AutoProfile.BaseComponent.ReviewStatus'},... % Stereotype property to qualify by
  'IncludeReferenceModels',true,... % Include components in referenced models
  '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'.

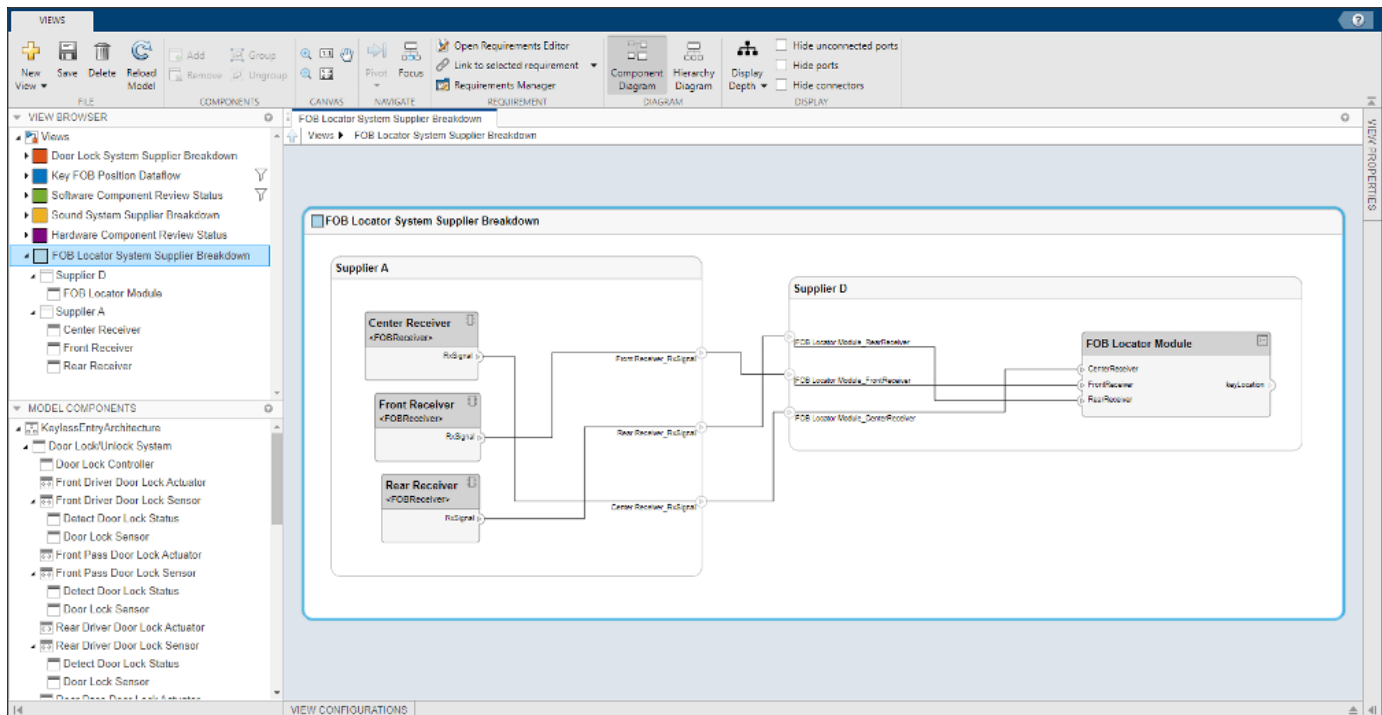
```
supplierA = fobSupplierView.Root.createSubGroup('Supplier A');
```

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 architecture design.	<p>You can use different types of views to represent the system:</p> <ul style="list-style-type: none"> • <i>Operational views</i> demonstrate how a system will be used and should be integrated with requirements analysis. • <i>Functional views</i> focus on what the system must do to operate. • <i>Physical views</i> show how the system is constructed and configured. <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"> • “Create Architecture Views Interactively” • “Create Architectural Views Programmatically”
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"> • <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. • <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. 	"Display Component Hierarchy and Architecture Hierarchy Using Views"

See Also

`openViews` | `createView` | `getView` | `deleteView` | `systemcomposer.view.ElementGroup`

Topics

"Create Architecture Views Interactively"

"Create Architectural Views Programmatically"

"Display Component Hierarchy and Architecture Hierarchy Using Views"

Introduced in R2021a

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 class inherits from the `systemcomposer.view.ViewElement` class.

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

Compatibility Considerations**systemcomposer.view.ViewArchitecture class has been removed***Errors starting in R2021a*

The `systemcomposer.view.ViewArchitecture` class is removed in R2021a with the introduction of new views APIs. 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”

Introduced in R2019b

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 class inherits from the `systemcomposer.view.BaseViewComponent` class.

Compatibility Considerations

systemcomposer.view.ViewComponent class has been removed

Errors starting in R2021a

The `systemcomposer.view.ViewComponent` class is removed in R2021a with the introduction of new views APIs. 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”

Introduced in R2019b

systemcomposer.view.ViewElement

(Removed) All view elements

Note The `systemcomposer.view.ViewElement` 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

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`

Compatibility Considerations

systemcomposer.view.ViewElement class has been removed

Errors starting in R2021a

The `systemcomposer.view.ViewElement` class is removed in R2021a with the introduction of new views APIs. 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”

Introduced in R2009b

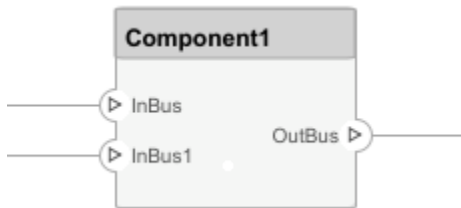
Blocks

Component

Add component to an 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.

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	“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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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"

See Also

Functions

`addComponent` | `addPort` | `connect`

Blocks

Reference Component | Variant Component | Adapter

Topics

"Compose Architecture Visually"

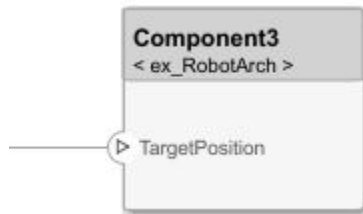
Introduced in R2019a

Reference Component

Link to an 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 Model** to save the contents of the component as an architecture model 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.
 - Select **Create Simulink Behavior** to create a new Simulink reference model and link to it.
 - Select **Link to Model** to link to a known model 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.

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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.	"Describe 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 a 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 at least one <code>Simulink.ConnectionElement</code> object.	Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.	"Specify Physical Interfaces on the 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 or Simulink behavior model.	A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.	<ul style="list-style-type: none"> • "Describe Component Behavior Using Simulink" • "Create Reference Architecture"
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"> • "Create Simulink Behavior Using Simulink Subsystem" • "Create a Simulink Subsystem Component"

Term	Definition	Application	More Information
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"> • “Implement Behaviors for Architecture Model Simulation” • “Describe Component Behavior Using Stateflow Charts”
sequence diagram	A sequence diagram is a behavior diagram that represents the interaction between structural elements of an architecture as a sequence of message exchanges.	You can use sequence diagrams to describe how the parts of a static system interact.	<ul style="list-style-type: none"> • “Describe System Behavior Using Sequence Diagrams” • “Use Sequence Diagrams with Architecture Models”

See Also

Functions

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

Blocks

Component | Variant Component | Adapter

Topics

“Describe Component Behavior Using Simulink”
 “Decompose and Reuse Components”
 “Describe Component Behavior Using Stateflow Charts”
 “Create Simulink Behavior Using Simulink Subsystem”
 “Simulate and Deploy Software Architectures”

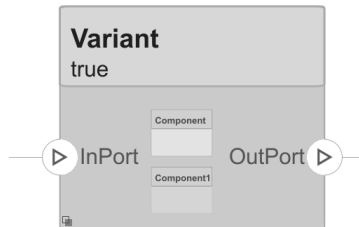
Introduced in R2019a

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**.

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. You can represent specific architectures using alternate views.	Different types of architectures describe different aspects of systems: <ul style="list-style-type: none"> • <i>Functional architecture</i> describes the flow of data in a system. • <i>Logical architecture</i> describes the intended operation of a system. • <i>Physical architecture</i> describes the platform or hardware in a system. 	“Compose Architecture Visually”
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"> • Extract the root-level architecture contained in the model. • Apply profiles. • Link interface data dictionaries. • Generate instances from model architecture. System Composer models are stored as SLX files.	“Create an 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 architecture 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.	"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"> • <i>Component ports</i> are interaction points on the component to other components. • <i>Architecture ports</i> are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model. 	"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 Condition" on page 1-531

See Also

Functions

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

Blocks

Component | Reference Component | Adapter

Topics

“Decompose and Reuse Components”

Introduced in R2019a

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.

To use an Adapter block:

- Insert an Adapter block between two ports with different interfaces. You can create mappings between interface elements on each port.
- Double-click the Adapter block to open the **Edit Interface Mappings : Interface Adapter** dialog. From here, you can create and edit mappings between input and output interfaces, and apply interface conversions: `UnitDelay` to break an algebraic loop or `RateTransition` to reconcile different sample time rates for reference models. For more information, see “Interface Adapter”.

Limitations

- When used for structural interface adaptations, the Adapter block uses bus element ports internally and, subsequently, only supports virtual buses.

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 Interface Editor. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.	<ul style="list-style-type: none"> • “Manage Interfaces with Data Dictionaries” • “Reference Data Dictionaries”
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.	“Create an Architecture Model with Interfaces and Requirement Links”
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"> • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components. 	<ul style="list-style-type: none"> • “Create Interfaces” • “Assign Interfaces to Ports”
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.	“Create Value Types as Interfaces”

Term	Definition	Application	More Information
owned interface	An owned interface is a locally defined 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 also act as a unit delay or rate transition. Use the Adapter block to implement an adapter.	With an adapter, you can perform functions on the Interface Adapter dialog: <ul style="list-style-type: none"> • Create and edit mappings between input and output interfaces. • Apply an interface conversion <code>UnitDelay</code> to break an algebraic loop. • Apply an interface conversion <code>RateTransition</code> to reconcile different sample time rates for reference models. 	<ul style="list-style-type: none"> • “Interface Adapter” • Adapter

See Also

Functions

connect

Blocks

Component | Reference Component | Variant Component

Topics

“Create Interfaces”

Introduced in R2019a

Sequence Viewer

Visualize messages, events, states, transitions, and functions

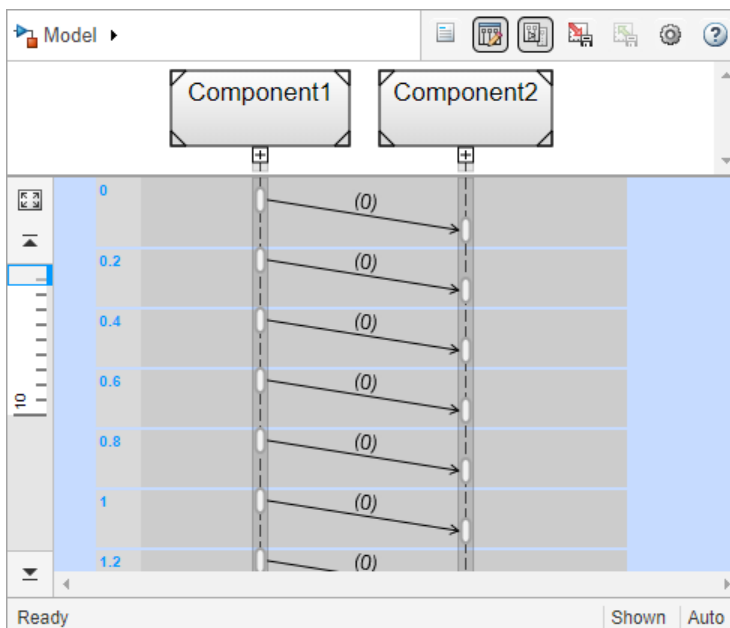
Description

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 “Use the Sequence Viewer to Visualize Messages, Events, and Entities”.

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**.
- “Use the Sequence Viewer to Visualize Messages, Events, and Entities”
 - “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'

See Also

Topics

“Use the Sequence Viewer to Visualize Messages, Events, and Entities”

“Simulink Messages Overview”

Introduced in R2020b