System Composer™

Reference

MATLAB® & SIMULINK®

R2021a

MathWorks®

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

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)

Contents

1

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

Variant choice names, specified as a cell array of character vectors. The length of `choices` must be the same as `labels`.

Data Types: `char`

`labels` — Variant choice labels
cell array of character vectors

Variant choice labels, specified as a cell array of character vectors. The length of `labels` must be the same as `choices`.

Data Types: `char`

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 1-417

See Also

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

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 cell array of names.

`components = addComponent(architecture,compNames,stereotypes)` applies stereotypes specified in the `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

Name of components, specified as a cell array of character vectors. The length of `compNames` must be the same as `stereotypes`.

Data Types: `char`

`stereotypes` — Stereotypes to apply to components
cell array of character vectors

Stereotypes to apply to components, specified as a cell array of character vectors. Each element is the qualified stereotype name for the corresponding component in the form `'<profile>.<stereotype>'`.

Data Types: `char`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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"
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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

Component | `addPort` | `connect`

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 then 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 editor to see it.

```
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 the component
character vector

Path to the component including the name of the top-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 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
`createView` | `deleteView` | `getView` | `openViews` | `systemcomposer.view.ElementGroup` | `systemcomposer.view.View`

Topics
"Create Architecture Views Interactively"
"Create Architectural Views Programmatically"

Introduced in R2019b

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'`

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• “Create Architecture Views Interactively” • “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.	“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 a Model Using Queries”

See Also

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

Topics
"Create Architecture Views Interactively"
"Create Architectural Views Programmatically"

Introduced in R2021a

addElement

Package: `systemcomposer.interface`

Add signal interface element

Syntax

```
element = addElement(interface,name)
element = addElement(interface,name,Name,Value)
```

Description

`element = addElement(interface,name)` adds an element to a signal interface with default properties.

`element = addElement(interface,name,Name,Value)` sets the properties of the element as specified in `Name,Value`.

Examples

Add an Interface and an Element

Add an interface `'newSignal'` to the interface dictionary of the model, and add an element `'newElement'` with type `'double'`.

```
arch = systemcomposer.createModel('newModel',true);
interface = addInterface(arch.InterfaceDictionary,'newSignal');
element = addElement(interface,'newElement','Type','double')
```

```
element = 
  SignalElement with properties:

      Interface: [1×1 systemcomposer.interface.SignalInterface]
           Name: 'newElement'
           Type: 'double'
     Dimensions: '1'
          Units: ''
     Complexity: 'real'
        Minimum: '[]'
        Maximum: '[]'
    Description: ''
           UUID: '2b47eaa6-191a-439a-ba2b-2bcc3209b912'
    ExternalUID: ''
```

Input Arguments

`interface` — New interface object
signal interface

New interface object, specified as a `systemcomposer.interface.SignalInterface` object.

`name` — Name of new element
character vector

Name of new element with a valid variable name, specified as a character vector.

Data Types: `char`

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` — Data type of element
valid data type character vector

Data type of element, specified as the comma-separated pair consisting of `'Type'` and a valid data type character vector.

Data Types: `char`

`Dimensions` — Dimensions of element
positive integer array

Dimensions of element, specified as the comma-separated pair consisting of `'Dimensions'` and a positive integer array. Each element of the array is the size of the element in the corresponding direction. A scalar integer indicates a scalar or vector element and a row vector with two integers indicates a matrix element.

Data Types: `double`

`Complexity` — Complexity of element
`'real'` | `'complex'`

Complexity of element, specified as the comma-separated pair `'Complexity'` and `'real'` if the element is purely real, or `'complex'` if an imaginary part is allowed.

Data Types: `char`

Output Arguments

`element` — New interface element object
signal element

New interface element object, returned as a `systemcomposer.interface.SignalElement` object.

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

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

Topics
"Define Interfaces"

Introduced in R2019a

addInterface

Package: `systemcomposer.interface`

Create named interface in interface dictionary

Syntax

```
interface = addInterface(dictionary,name)
interface = addInterface(dictionary,name,'SimulinkBus',busObject)
```

Description

`interface = addInterface(dictionary,name)` adds a named interface to a specified interface dictionary.

`interface = addInterface(dictionary,name,'SimulinkBus',busObject)` constructs an interface that mirrors an existing Simulink® bus object.

Examples

Add an Interface

Add an interface `'newInterface'` to the specified data dictionary and then create a model, link the dictionary, and view the interface editor.

Create a data dictionary and add an interface.

```
dictionary = systemcomposer.createDictionary('new_dictionary.sldd');
interface = addInterface(dictionary,'newInterface')
```

Create a new model, link the data dictionary, and open the interface editor.

```
arch = systemcomposer.createModel('newModel',true);
linkDictionary(arch,'new_dictionary.sldd');
```

Add a Simulink Bus Mirrored Interface

Add a named interface that mirrors an existing Simulink bus object to a specified dictionary. Create a model, link the dictionary, and view the interface editor.

Create a dictionary, create a Simulink bus object, populate the bus object with two elements, and add the named 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. This is 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 interface

character vector

Name of new interface, specified as a character vector.

Data Types: `char`

`busObject` — Simulink bus object that new interface mirrors

bus object

Simulink bus object that new interface mirrors where the interface is already defined, specified as a Simulink bus object.

Output Arguments

`interface` — New interface object

signal interface object

New interface object, returned as a `systemcomposer.interface.SignalInterface` object.

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

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

Topics

"Define Interfaces"

Introduced in R2019a

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.

`ports = addPort(architecture,portNames,portTypes,stereotypes)` also applies stereotypes to a set of 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: [1×1 systemcomposer.arch.Architecture]
                   Name: 'NewCompPort'
              Direction: Input
          InterfaceName: ''
              Interface: [0×0 systemcomposer.interface.SignalInterface]
             Connectors: [0×0 systemcomposer.arch.Connector]
              Connected: 0
                  Model: [1×1 systemcomposer.arch.Model]
         SimulinkHandle: 52.0001
    SimulinkModelHandle: 49.0001
                   UUID: '98070dc5-1738-4dbf-b9b2-4fc781e7992c'
            ExternalUID: ''
```

Input Arguments

`architecture` — Component architecture
architecture object

Component architecture, specified as a `systemcomposer.arch.Architecture` object. `addPort` adds ports to the architecture of a component. Use `<component>.Architecture` to access the architecture of a component.

`portNames` — Names of ports
cell array of character vectors

Names of ports, specified as a cell array of character vectors. 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`

`portTypes` — Port directions
cell array of character vectors

Port directions, specified as a cell array of character vectors. A port direction can be either `'in'` or `'out'`.

Data Types: `char`

`stereotypes` — Stereotypes to apply to components
array of stereotype objects

Stereotypes to apply to components, 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

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

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

Name of property unique within the stereotype, specified as a character vector.

Data Types: `char`

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

Units of the property value, specified as a character vector. 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`

`DefaultValue` — Default value
character vector

Default value of this property, specified as a character vector that can be evaluated depending on the `'Type'`.

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`getProperty` | `removeProperty` | `setProperty`

Topics

"Define Profiles and Stereotypes"
"Set Properties for Analysis"

Introduced in R2019a

addReference

Package: `systemcomposer.interface`

Add reference to dictionary

Syntax

`addReference(dictionary,reference)`

Description

`addReference(dictionary,reference)` 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

Referenced dictionary, specified as a character vector of the name of the referenced dictionary with the `.sldd` extension.

Example: `'ReferenceDictionary.sldd'`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	“Assign Interfaces to Ports”
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• “Save, Link, and Delete Interfaces” • “Reference Data Dictionaries”

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`createDictionary` | `linkDictionary` | `openDictionary` | `removeReference` | `saveToDictionary` | `unlinkDictionary`

Topics

"Save, Link, and Delete Interfaces"
"Reference 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)` specifies the properties of the stereotype.

Examples

Add Component Stereotype

Add a component stereotype to the profile.

```
profile = systemcomposer.profile.Profile.createProfile('LatencyProfile');
stereotype = addStereotype(profile,'electricalComponent','AppliesTo','Component')
```

```
stereotype = 

  Stereotype with properties:

                    Name: 'electricalComponent'
             Description: ''
                  Parent: [0×0 systemcomposer.profile.Stereotype]
               AppliesTo: 'Component'
                Abstract: 0
                    Icon: 'default'
    ComponentHeaderColor: [210 210 210 255]
      ConnectorLineColor: [168 168 168 255]
      ConnectorLineStyle: 'Default'
      FullyQualifiedName: 'LatencyProfile.electricalComponent'
                 Profile: [1×1 systemcomposer.profile.Profile]
         OwnedProperties: [0×0 systemcomposer.profile.Property]
              Properties: [0×0 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

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

Data Types: `char`

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')`

`Name, Value` — Stereotype properties and values
positive integer array

See `systemcomposer.profile.Stereotype` for stereotype properties and values.

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"

Term	Definition	Application	More Information
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

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

Topics

“Create a Profile and Add Stereotypes”

Introduced in R2019a

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

Names of variant components, specified as a cell array of character vectors.

Data Types: `char`

`position` — Vector that specifies location of top corner and bottom corner of component
1x4 array

Vector that specifies location of top corner and bottom corner of component, specified as a 1x4 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 1-417

See Also

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

Topics

"Components"

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 and the target element.

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 to create allocations in, 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`, or `systemcomposer.arch.Connector` 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`, or `systemcomposer.arch.Connector` 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 a 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` | `deallocate` | `destroy` | `getAllocatedFrom` | `getAllocatedTo` | `getAllocation` | `getScenario`

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` object that the `find` method and the `createView` method 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 =

  1×1 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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

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

Name of profile, specified as a character vector.

Example: `'SystemProfile'`

Data Types: `char`

More About

Definitions

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>architecture</td><td>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.</td><td>Different types of architectures describe different aspects of systems:
Functional architecture describes the flow of data in a system.
Logical architecture describes the intended operation of a system.
Physical architecture describes the platform or hardware in a system.</td><td>“Compose Architecture Visually”</td></tr>
<tr><td>model</td><td>A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.</td><td>Perform operations on a model:
Extract the root level architecture contained in the model.
Apply profiles.
Link interface data dictionaries.
Generate instances from model architecture.
System Composer models are stored as <code>.slx</code> files.</td><td>“Create an Architecture Model”</td></tr>
<tr><td>component</td><td>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.</td><td>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts.</td><td>“Components”</td></tr>
</table>

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`createProfile` | `removeProfile`

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()

stereotypes = getStereotypes(comp)
```

```
stereotypes =

  1×1 cell array
```

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

Input Arguments

element — Model element

architecture object | component object | port object | connector object | signal 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, or systemcomposer.interface.SignalInterface object.

stereotype — Name of stereotype

character vector

Name of stereotype, specified as a character vector in the form '<profile>.<stereotype>'. The profile must already be applied to the model.

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also
`batchApplyStereotype` | `getStereotypes` | `removeStereotype`

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 of `'Component'`, `'Port'`, `'Connector'`, or `'Instance'`. The stereotype must be applicable for this element type.

Data Types: `char`

`stereotype` — Stereotype to apply
character vector

Stereotype to apply, specified as a character vector in the form `'<profile>.<stereotype>'`. The stereotype must be applicable to components.

Data Types: `char`

`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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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"
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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`applyStereotype` | `getStereotypes` | `removeStereotype`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

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

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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` | `save`

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. 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 a 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`

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 a 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`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

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

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 `srcComponent` to the unconnected input ports of `destComponent` based on matching port names, and returns a handle to the connector.

`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.

`connectors = connect(srcPort,destPort,stereotype)` connects a source port and a destination port and applies a stereotype to the connector.

`connectors = connect( ___ ,Name,Value)` specifies options using one or more name-value pair 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 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 destination architecture port interface element to 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 and 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` — Interface and underlying structural definition of model or component
architecture object

Interface and underlying structural definition of model or component, specified as a `systemcomposer.arch.Architecture` object.

`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

Stereotype to apply to the connection, specified as a fully-qualified name in the form `'<profile>.<stereotype>'`.

Data Types: `char`

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

Stereotype — Option to apply stereotype to connector
character vector

Option to apply stereotype to connector, specified as the comma-separated pair consisting of `'Stereotype'` and a fully-qualified name in the form `'<profile>.<stereotype>'`.

This name-value pair only applies when connecting components.

Example: `conns = connect(srcComp,destComp,'Stereotype','GeneralProfile.ConnStereotype')`

Data Types: `char`

Rule — Option to specify rule for connections
`'name'` (default) | `'interface'`

Option to specify rule for connections, specified as the comma-separated pair consisting of `'Rule'` and either `'name'` based on the name of ports or `'interface'` based on the interface name on ports.

This name-value pair only applies when connecting components.

Example: `conns = connect([srcComp1,srcComp2], [destComp1,destComp2],'Rule','interface')`

Data Types: `char`

MultipleOutputConnectors — Option to allow multiple destination components
`false` or 0 (default) | `true` or 1

ption to allow multiple destination components for the same source component, specified as the comma-separated pair consisting of `'MultipleOutputConnectors'` and a logical 1 (`true`) or 0 (`false`).

This name-value pair only applies when connecting components.

Example: `conns = connect(srcComp, [destComp1,destComp2],'MultipleOutputConnectors',true)`

Data Types: `logical`

`SourceElement` — Option to select source element for connection
character vector

Option to select source element for connection, specified as the comma-separated pair consisting of `'SourceElement'` and a character vector of the name of the signal element.

This name-value pair only applies when connecting ports.

Example: `conns = connect(archSrcPort,compDestPort,'SourceElement','x')`

Data Types: `char`

`DestinationElement` — Option to select destination element for connection
character vector

Option to select destination element for connection, specified as the comma-separated pair consisting of `'DestinationElement'` and a character vector of the name of the signal element.

This name-value pair only applies when connecting ports.

Example: `conns = connect(compSrcPort,archDestPort,'DestinationElement','x')`

Data Types: `char`

`Routing` — Option to specify type of automatic line routing
`'smart'` (default) | `'on'` | `'off'`

Option to specify type of automatic line routing, specified as the comma-separated pair consisting of `'Routing'` and one of the following:

- `'smart'` for automatic line routing that takes the best advantage of the blank spaces on the canvas and avoids overlapping other lines and labels.
- `'on'` for automatic line routing.
- `'off'` for no automatic line routing.

Example: `conns = connect(srcPort,destPort,'Routing','on')`

Data Types: `char`

Output Arguments

`connectors` — Created connections
array of connections

Created connections, returned as an array of `systemcomposer.arch.Connector` 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 | `addComponent` | `addElement` | `addInterface` | `addPort` | `createModel` | `getDestinationElement` | `getPort` | `getSourceElement` | `openModel` | `setInterface`

Topics

“Connections”
“Build an Architecture Model from Command Line”

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

Name of allocation set, specified as a character vector.

Example: `'MyNewAllocation'`

Data Types: `char`

`sourceModel` — Source model for allocation
model object | character vector

Source model for allocation, specified as a `systemcomposer.arch.Model` object or the name of a model as a character vector.

`targetModel` — Target model for allocation
model object | character vector

Target model for allocation, specified as a `systemcomposer.arch.Model` object or the name of a model as a character vector.

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 a 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

`closeAll` | `load` | `open`

Topics
“Create and Manage Allocations”

Introduced in R2020b

createAnonymousInterface

Package: systemcomposer.arch

Create and set anonymous interface for port

Syntax

`interface = createAnonymousInterface(port)`

Description

`interface = createAnonymousInterface(port)` creates and sets an anonymous interface for a port.

Examples

Add Port to Architecture and Set Anonymous Interface

Create a model, get the root architecture, add a component, and add a port. Set an anonymous interface for the port.

```
model = systemcomposer.createModel('archModel',true);
rootArch = get(model,'Architecture');
newComponent = addComponent(rootArch,'NewComponent');
newPort = addPort(newComponent.Architecture,'NewCompPort','in');
interface = createAnonymousInterface(newPort)
```

```
interface = 

  SignalInterface with properties:

     Dictionary: []
           Name: ''
       Elements: [1×1 systemcomposer.interface.SignalElement]
          Model: [1×1 systemcomposer.arch.Model]
           UUID: '37046ccd-7cf5-4b2b-886a-10990bb3553e'
    ExternalUID: ''
```

Input Arguments

port — Port
port object

Port, specified as a `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` object.

Output Arguments

`interface` — Signal interface
signal interface object

Signal interface, returned as a `systemcomposer.interface.SignalInterface` 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”

Term	Definition	Application	More Information
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

Component | `systemcomposer.arch.ArchitecturePort` | `systemcomposer.arch.ComponentPort`

Topics

"Define Interfaces"

Introduced in R2019a

systemcomposer.createDictionary

Create data dictionary

Syntax

`dict_id = systemcomposer.createDictionary(dictionaryName)`

Description

`dict_id = systemcomposer.createDictionary(dictionaryName)` creates a new Simulink data dictionary to hold interfaces and returns the `systemcomposer.interface.Dictionary` object.

Examples

Create New Dictionary

```
dict_id = systemcomposer.createDictionary('new_dictionary.sldd')
```

Input Arguments

`dictionaryName` — Name of new data dictionary
character vector

Name of new data dictionary, specified as a character vector. The name must include the `.sldd` extension.

Example: `'new_dictionary.sldd'`

Data Types: `char`

Output Arguments

`dict_id` — 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"

Term	Definition	Application	More Information
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addReference` | `linkDictionary` | `openDictionary` | `removeReference` | `saveToDictionary` | `unlinkDictionary`

Topics

"Save, Link, and Delete Interfaces"
"Reference Data Dictionaries"

Introduced in R2019a

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: [1×1 systemcomposer.arch.Architecture]
       SimulinkHandle: 2.0005
                Views: [0×0 systemcomposer.view.ViewArchitecture]
             Profiles: [0×0 systemcomposer.profile.Profile]
  InterfaceDictionary: [1×1 systemcomposer.interface.Dictionary]
```

Input Arguments

`modelName` — Name of new model
character vector

Name of new model, specified as a character vector.

Example: `'model_name'`

Data Types: `char`

`openFlag` — Whether to open model
`false` or 0 (default) | `true` or 1

Whether to open model upon creation, specified as a logical 1 (`true`) to open the model or 0 (`false`) to not open the model.

Data Types: `logical`

`modelType` — Type of model
`'Architecture'` (default) | `'SoftwareArchitecture'`

Type of model to create, specified as a character vector `'Architecture'` for an architecture model or `'SoftwareArchitecture'` for a software architecture model.

Data Types: `char`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 a description of 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.	“Author 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.	“Simulate and Deploy Software Architectures”
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”

See Also

`loadModel` | `open` | `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

Name of new profile, specified as a character vector.

Example: `'LatencyProfile'`

Data Types: `char`

`dirPath` — Directory path
character vector

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

Example: `'C:\Temp\MATLAB'`

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

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

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`.

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

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

Example: `'Scenario 2'`

Data Types: `char`

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 a 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`

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 with the same interface as the component and links the component to the new model. The component must have no children.

Examples

Create Simulink Model and Link

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

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

Model name of the Simulink model created by this function, specified as a character vector.

Example: `'Robot'`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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.	• "Implement Component Behavior in Simulink" • "Create a Reference Architecture"
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 an architectural component using state machines.	"Add Stateflow Chart Behavior to Architecture Component"
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.	• "Define Sequence Diagrams" • "Use Sequence Diagrams in the Views Gallery"

Term	Definition	Application	More Information
software architecture	A software architecture is a specialization of an architecture for software-based systems, including a description of 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.	"Author 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.	"Simulate and Deploy Software Architectures"
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"

See Also

Reference Component | `createStateflowChartBehavior` | `extractArchitectureFromSimulink` | `inlineComponent` | `isReference` | `linkToModel` | `saveAsModel`

Topics

"Implement Component Behavior in Simulink"
"Decompose and Reuse Components"
"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. The connections, interfaces, requirement links, and stereotypes are preserved. The component must have no sub-components and must not already be linked to a model.

Examples

Add Stateflow Chart Behavior to Component

Add a 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 — Architecture component
component object

Architecture component with no sub-components, 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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.	• “Implement Component Behavior in Simulink” • “Create a Reference Architecture”
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 an architectural component using state machines.	“Add Stateflow Chart Behavior to Architecture Component”
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.	• “Define Sequence Diagrams” • “Use Sequence Diagrams in the Views Gallery”

See Also

`createSimulinkBehavior` | `extractArchitectureFromSimulink` | `inlineComponent` | `isReference` | `linkToModel` | `saveAsModel`

Topics

"Add Stateflow Chart Behavior to Architecture Component"

Introduced in R2021a

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: [0×0 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

Name of subgroup, specified as a character vector.

Data Types: `char`

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

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

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

Name of new view, specified as a character vector.

Example: `'All Components Grouped by Review Status'`

Data Types: `char`

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 properties

Grouping criteria, specified as a comma-separating pair consisting of `'GroupBy'` and a cell array of properties 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'}`

`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

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` — 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

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

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

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

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

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

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 a source and a target element.

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

`sourceElement` — Source element to delete allocation

element object

Source element 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`, or `systemcomposer.arch.Connector` object.

`targetElement` — Target element to delete allocation
element object

Target element 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`, or `systemcomposer.arch.Connector` 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 a 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` | `createAllocationSet` | `destroy` | `getAllocatedFrom` | `getAllocatedTo` | `getAllocation` | `getScenario`

Topics

“Create and Manage Allocations”

Introduced in R2020b

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	“Analyze Architecture”
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	“Create a Model Instance for Analysis”

See Also

`instantiate` | `loadInstance` | `refresh` | `save` | `systemcomposer.analysis.Instance` | `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 a set with a given 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

Name of allocation scenario to be deleted, specified as a character vector.

Example: `'Scenario 1'`

Data Types: `char`

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 a 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` | `getScenario`

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

Name of subgroup, specified as a character vector.

Example: `'MyGroup'`

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

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

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'`.

```
scKeylessEntrySystem
model = systemcomposer.loadModel('KeylessEntryArchitecture');
view = model.createView('NewView');
```

Open the Architecture Views Gallery to see `'NewView'`.

```
model.openViews
```

Delete the view and see that it has been deleted.

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

Input Arguments

`model` — Model
model object

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

`viewName` — Name of view
character vector

Name of view, specified as a character vector.

Example: `'NewView'`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• “Create Architecture Views Interactively” • “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.	“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 a Model Using Queries”

See Also

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

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 | component port object | architecture port object | connector object | signal interface object | signal 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.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.Connector`
- `systemcomposer.interface.SignalInterface`
- `systemcomposer.interface.SignalElement`
- `systemcomposer.profile.Property`
- `systemcomposer.view.View`
- `systemcomposer.view.ElementGroup`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	“Assign Interfaces to Ports”

Term	Definition	Application	More Information
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"

Term	Definition	Application	More Information
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• “Create Architecture Views Interactively” • “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.	“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 a Model Using Queries”

.

See Also

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

Introduced in R2019a

destroy

Package: systemcomposer.allocation

Remove allocation scenario

Syntax

`destroy(allocScenario)`

Description

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

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.

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 a 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` | `createAllocationSet` | `createScenario` | `deallocate` | `deleteScenario` | `getScenario`

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 a 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

Name of profile to select, specified as a character vector.

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

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

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

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 requirements to be imported into MATLAB® tables. The exported tables have prescribed formats to specify model element relationships, stereotypes, and properties.

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`.

```
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]
```

Input Arguments

`modelName` — Name of model to be exported
character vector

Name of model to be exported, specified as a character vector.

Example: `'exMobileRobot'`

Data Types: `char`

Output Arguments

`exportedSet` — Model tables
structure

Model tables, returned as a structure containing tables `components`, `ports`, `connections`, `portInterfaces`, and `requirementLinks`.

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”

Term	Definition	Application	More Information
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

Term	Definition	Application	More Information
requirements	A collection of statements describing the desired behavior and characteristics of a system. Requirements ensure system design integrity and are achievable, verifiable, unambiguous, and consistent with each other. Each level of design should have appropriate requirements.	To enhance traceability of requirements, link system, functional, customer, performance, or design requirements to components and ports. Link requirements to each other to represent derived or allocated requirements. Manage requirements from the requirements perspective on an architecture model or through custom views. Assign test cases to requirements.	• “Link and Trace Requirements” • “Manage Requirements” • “Update Reference Requirement Links from Imported File” on page 1-477

See Also

`importModel`

Topics

“Import and Export Architecture Models”

Introduced in R2019a

systemcomposer.extractArchitectureFromSimulink

Extract architecture from Simulink model

Syntax

`systemcomposer.extractArchitectureFromSimulink(model,name)`

Description

`systemcomposer.extractArchitectureFromSimulink(model,name)` exports the Simulink model `model` to an architecture model `architectureModelName` and saves it in the current directory.

Examples

Extract Architecture from Example Model

Extract architecture from a model with subsystem and variant architecture.

```
ex_modeling_variants
systemcomposer.extractArchitectureFromSimulink('ex_modeling_variants','archModel')
```

Input Arguments

`model` — Simulink model name
character vector

Simulink model name from which to extract the architecture, specified as a character vector. The model must be on the path.

Example: `'ex_modeling_variants'`

Data Types: `char`

name — Architecture model name
character vector

Architecture model name, specified as a character vector. This model is saved in the current directory.

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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` | `linkToModel` | `saveAsModel`

Topics

"Extract Architecture from Simulink Model"

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 pair arguments.

`[paths, elements] = find( ___ )` returns the component `elements` and their `paths` that satisfy the `constraint` query. 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` that satisfy the `constraint` query, with additional options specified by one or more name-value pair 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 that satisfy the `constraint` query, with additional options specified by one or more name-value pair 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 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 System Composer 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: 2.0027
    SimulinkModelHandle: 0.0027
                   UUID: '5b007388-c938-4dcf-bbaf-81efefb6e562'
            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 the `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
character vector

Root architecture of model, specified as a character vector.

Data Types: `char`

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: `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 the comma-separated pair consisting of `'Recurse'` and a numeric or logical 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 to not include referenced architectures, specified as the comma-separated pair consisting of `'IncludeReferenceModels'` and a logical 0 (`false`) to not include 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 the comma-separated pair consisting of `'ElementType'` and `'Component'` to select components to satisfy the query, `'Port'` to select ports to satisfy the query, or `'Connector'` to select 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

`createView` | `systemcomposer.query.Constraint`

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 model.

```
scExampleSmallUAV
stereotype = systemcomposer.profile.Stereotype.find('UAVComponent.OnboardElement')
```

```
stereotype = 

  Stereotype with properties:

                   Name: 'OnboardElement'
            Description: 'Represents the base component of UAVComponent'
                 Parent: [0×0 systemcomposer.profile.Stereotype]
              AppliesTo: 'Component'
               Abstract: 0
                   Icon: 'network'
   ComponentHeaderColor: [210 210 210 255]
     ConnectorLineColor: [168 168 168 255]
     ConnectorLineStyle: 'Default'
     FullyQualifiedName: 'UAVComponent.OnboardElement'
                Profile: [1×1 systemcomposer.profile.Profile]
        OwnedProperties: [1×3 systemcomposer.profile.Property]
             Properties: [1×3 systemcomposer.profile.Property]
```

Input Arguments

name — Name of stereotype
character vector

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

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`setDefaultComponentStereotype` | `setDefaultConnectorStereotype` | `setDefaultPortStereotype` | `systemcomposer.profile.Stereotype`

Introduced in R2019a

systemcomposer.profile.Profile.find

Find profile by name

Syntax

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

Description

`profile = systemcomposer.profile.Profile.find(name)` finds a profile by 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: [1×5 systemcomposer.profile.Stereotype]
```

Input Arguments

name — Name of profile
character vector

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

Example: `'LatencyProfile'`

Data Types: `char`

Output Arguments

`profile` — Found profile
profile object

Found 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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

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

Topics
“Define Profiles and Stereotypes”

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.

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

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

Example: `'Scenario 1'`

Data Types: `char`

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 a 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

`closeAll` | `load` | `save`

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: [1×1 systemcomposer.arch.Architecture]
                   Name: 'Choice2'
                 Parent: [1×1 systemcomposer.arch.Architecture]
                  Ports: [0×0 systemcomposer.arch.ComponentPort]
             OwnedPorts: [0×0 systemcomposer.arch.ComponentPort]
      OwnedArchitecture: [1×1 systemcomposer.arch.Architecture]
               Position: [15 15 65 65]
                  Model: [1×1 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 1-417

See Also

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

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 a target is allocated from.

Examples

Create Allocation Set, Allocate Elements, and Get Allocated From

```
% 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` — Source 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`, or `systemcomposer.arch.Connector` object.

Output Arguments

`sourceElements` — Target elements
array of element objects

Source elements allocated from that are allocated to the specified target element, 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`, or `systemcomposer.arch.Connector` 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 a 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` | `getAllocatedTo`

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 the specified source element is allocated to.

Examples

Create Allocation Set, Allocate Elements, and Get Allocated To

```
% 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`, or `systemcomposer.arch.Connector` object.

Output Arguments

`targetElements` — Target elements
array of element objects

Target elements that are allocated to, 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`, or `systemcomposer.arch.Connector` 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 a 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` | `getAllocatedFrom`

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 and target element.

Examples

Create Allocation Set, Allocate, and Get Allocation

```
% 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.ComponentPort, systemcomposer.arch.ArchitecturePort, or systemcomposer.arch.Connector 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.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, or `systemcomposer.arch.Connector` 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 a 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` | `getAllocatedFrom` | `getAllocatedTo`

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: [1×1 systemcomposer.arch.Architecture]
                   Name: 'Choice1'
                 Parent: [1×1 systemcomposer.arch.Architecture]
                  Ports: [0×0 systemcomposer.arch.ComponentPort]
             OwnedPorts: [0×0 systemcomposer.arch.ComponentPort]
      OwnedArchitecture: [1×1 systemcomposer.arch.Architecture]
               Position: [15 15 65 65]
                  Model: [1×1 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 1-417

See Also

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

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 1-417

See Also

Variant Component | `addVariantComponent` | `makeVariant` | `setActiveChoice` | `setCondition`

Topics
"Create Variants"

Introduced in R2019a

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('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, open the profile editor, then get the default stereotype.

```
profile.setDefaultStereotype('NodeLatency');

systemcomposer.profile.editor(profile)

default = getDefaultStereotype(profile)
```

```
default =

  Stereotype with properties:

                   Name: 'NodeLatency'
            Description: ''
                 Parent: [0×0 systemcomposer.profile.Stereotype]
              AppliesTo: 'Component'
               Abstract: 0
                   Icon: 'default'
   ComponentHeaderColor: [210 210 210 255]
     ConnectorLineColor: [168 168 168 255]
     ConnectorLineStyle: 'Default'
      FullyQualifiedName: 'LatencyProfile.NodeLatency'
                Profile: [1×1 systemcomposer.profile.Profile]
```

```
        OwnedProperties: [1×1 systemcomposer.profile.Property]
             Properties: [1×1 systemcomposer.profile.Property]
```

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

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

Topics
"Create a Profile and Add Stereotypes"

Introduced in R2019a

getDestinationElement

Package: `systemcomposer.arch`

Gets signal interface elements selected on destination port for connection

Syntax

`selectedElems = getDestinationElement(connector)`

Description

`selectedElems = getDestinationElement(connector)` gets the selected signal interface 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 =

  1×1 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 interface element names
character vector

Selected interface 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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"
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"

Term	Definition	Application	More Information
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	“Assign Interfaces to Ports”
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• “Save, Link, and Delete Interfaces” • “Reference Data Dictionaries”
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	“Interface Adapter”

See Also

Component | `addComponent` | `addElement` | `addInterface` | `addPort` | `connect` | `createModel` | `getPort` | `getSourceElement` | `setInterface` | `systemcomposer.arch.Connector`

Topics

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

Introduced in R2020b

getElement

Package: systemcomposer.interface

Get object for signal interface element

Syntax

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

Description

`element = getElement(interface,elementName)` gets the object for an element in a signal interface.

Examples

Get Object for Named Element

Add an interface `'newSignal'` to the interface dictionary of the model, and add an element `'newElement'` with type `'double'`. Then get the object for the element.

```
arch = systemcomposer.createModel('newModel',true);
interface = addInterface(arch.InterfaceDictionary,'newSignal');
addElement(interface,'newElement','Type','double');
element = getElement(interface,'newElement')
```

```
element =
  SignalElement with properties:

      Interface: [1×1 systemcomposer.interface.SignalInterface]
           Name: 'newElement'
           Type: 'double'
     Dimensions: '1'
          Units: ''
     Complexity: 'real'
        Minimum: '[]'
        Maximum: '[]'
    Description: ''
           UUID: 'f42c8166-e4ad-4488-926a-293050016e1a'
    ExternalUID: ''
```

Input Arguments

interface — Interface
signal interface object

Interface containing elements to be identified, specified as a `systemcomposer.interface.SignalInterface` object.

`elementName` — Name of interface element
character vector

Name of interface element to be identified, specified as a character vector.

Data Types: `char`

Output Arguments

`element` — Interface element
signal element object

Interface element, returned as a `systemcomposer.interface.SignalElement` object.

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	“Assign Interfaces to Ports”
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• “Save, Link, and Delete Interfaces” • “Reference Data Dictionaries”

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

Adapter | `addElement` | `getDestinationElement` | `getInterface` | `getSourceElement` | `removeElement`

Topics

"Define Interfaces"

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 | signal 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, or systemcomposer.interface.SignalInterface object.

property — Property name
character vector

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

Data Types: char

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`getProperty` | `getPropertyValue` | `getStereotypeProperties` | `setProperty`

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 Interface and Get Interface

Add an interface `'newInterface'` to the interface dictionary of the model. Obtain the interface object. Confirm by opening the interface editor.

```
arch = systemcomposer.createModel('newModel',true);
addInterface(arch.InterfaceDictionary,'newInterface');
interface = getInterface(arch.InterfaceDictionary,'newInterface')
```

```
interface =
  SignalInterface with properties:
     Dictionary: [1×1 systemcomposer.interface.Dictionary]
           Name: 'newInterface'
       Elements: [0×0 systemcomposer.interface.SignalElement]
           UUID: '438b5004-6cab-40eb-955b-37e0df5a914f'
    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

Name of interface, specified as a character vector.

Data Types: `char`

Output Arguments

`interface` — Object for named interface
signal interface object

Object for named interface, returned as a `systemcomposer.interface.SignalInterface` object.

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>adapter</td><td>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.</td><td>With an adapter, you can perform three functions on the Interface Adapter dialog:
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.</td><td>“Interface Adapter”</td></tr>
</table>

See Also

Adapter | `addElement` | `addInterface` | `getInterfaceNames` | `removeElement`

Topics

“Define Interfaces”

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 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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

Adapter | `addInterface` | `getInterface` | `removeInterface`

Topics

"Define Interfaces"

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 with a specified name.

Examples

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);
```

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

Name of port to find, specified as a character vector.

Data Types: `char`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 | `addComponent` | `addElement` | `addPort` | `connect` | `createModel` | `getElement` | `getInterface` | `getPort` | `removeElement`

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>'`.

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` — Architecture model element
architecture object | component object | port object | connector object

Architecture model element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, or `systemcomposer.arch.Connector` object.

`propertyName` — Name of property
character vector

Name of property, specified as a character vector in the form `'<profile>.<stereotype>.<property>'`.

Data Types: `char`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also
`addProperty` | `removeProperty` | `setProperty`

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 | signal interface object

Model element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.interface.SignalInterface` object.

`property` — Property name
character vector

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

Data Types: `char`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

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

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 this allocation set `allocSet` with the given `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

Name of allocation scenario, specified as a character vector.

Example: `'Scenario 1'`

Data Types: `char`

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 a 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`

Topics
“Create and Manage Allocations”

Introduced in R2020b

getSourceElement

Package: `systemcomposer.arch`

Gets signal interface elements selected on source port for connection

Syntax

`selectedElems = getSourceElement(connector)`

Description

`selectedElems = getSourceElement(connector)` gets the selected signal interface 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 =

  1×1 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 interface element names
character vector

Selected interface 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"

Term	Definition	Application	More Information
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

Component | `addComponent` | `addElement` | `addInterface` | `addPort` | `connect` | `createModel` | `getDestinationElement` | `getPort` | `setInterface` | `systemcomposer.arch.Connector`

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('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 editor. Get the stereotype `'ConnectorLatency'` in the profile.

```
systemcomposer.profile.editor()

stereotype = getStereotype(profile,'ConnectorLatency')
```

```
stereotype = 

  Stereotype with properties:

                  Name: 'ConnectorLatency'
           Description: ''
                Parent: [1×1 systemcomposer.profile.Stereotype]
             AppliesTo: {}
              Abstract: 0
                  Icon: 'default'
    ComponentHeaderColor: [210 210 210 255]
      ConnectorLineColor: [168 168 168 255]
      ConnectorLineStyle: 'Default'
      FullyQualifiedName: 'LatencyProfile.ConnectorLatency'
```

```
          Profile: [1×1 systemcomposer.profile.Profile]
  OwnedProperties: [1×2 systemcomposer.profile.Property]
       Properties: [1×4 systemcomposer.profile.Property]
```

Input Arguments

`profile` — Profile
profile object

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

`name` — Name of stereotype
character vector

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

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”

Term	Definition	Application	More Information
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`addStereotype` | `removeStereotype` | `systemcomposer.profile.Profile`

Topics

"Create a Profile and Add Stereotypes"

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 = 

    1×2 string array

    "LatencyProfile.electricalComponent.latency"    "LatencyProfile.electricalComponent.mass"
```

Input Arguments

`archElement` — Model element architecture

architecture object | architecture port object | connector object | signal interface object

Model element architecture, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.interface.SignalInterface` 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`getEvaluatedPropertyValue` | `getProperty` | `getPropertyValue` | `setProperty`

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 =

  1×1 cell array
```

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

Input Arguments

`element` — Model element
architecture object | component object | port object | connector object

Model element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, or `systemcomposer.arch.Connector` 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`applyStereotype` | `batchApplyStereotype` | `removeStereotype`

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: [0×0 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

Name of subgroup, specified as a character vector.

Data Types: `char`

Output Arguments

`subGroup` — Subgroup
element group object

Subgroup, returned as a `systemcomposer.view.ElementGroup` object.

More About

Definitions

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>view</td><td>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. Construct views by pulling in elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architecture design.</td><td>You can use different types of views to represent the system:
• Operational views demonstrate how a system will be used and should be well integrated with requirements analysis.
• Functional views focus on what the system must do to operate.
• Physical views show how the system is constructed and configured.
A viewpoint represents a stakeholder perspective that specifies the contents of the view.</td><td>• “Create Architecture Views Interactively”
• “Modeling System Architecture of Keyless Entry System”</td></tr>
<tr><td>element group</td><td>An element group is a grouping of components in a view.</td><td>Use element groups to programmatically populate a view.</td><td>“Create Architectural Views Programmatically”</td></tr>
<tr><td>query</td><td>A query is a specification that describes certain constraints or criteria to be satisfied by model elements.</td><td>Use queries to search elements with constraint criteria and to filter views.</td><td>“Find Elements in a Model Using Queries”</td></tr>
</table>

See Also

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

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` of the `instance` and assigns it to `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 by a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

`property` — Property
character vector

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

Data Types: `char`

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	“Analyze Architecture”
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

`hasValue` | `setValue` | `systemcomposer.analysis.Instance`

Topics

“Write Analysis Function”

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: [1×1 systemcomposer.view.ElementGroup]
                    Model: [1×1 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 new view
character vector

Name of new view, specified as a character vector.

Example: `'NewView'`

Data Types: `char`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

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

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` object that the `find` method and the `createView` method use to select architecture elements with an interface that satisfies the given 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 =

  10×1 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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries”

See Also

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

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` object that the `find` method and the `createView` method use to select architecture elements with an interface element that satisfies the given 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 =

  4×1 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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• “Create Architecture Views Interactively” • “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.	“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 a Model Using Queries”

See Also

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

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` object that the `find` method and the `createView` method use to select architecture elements with a port that satisfies the given 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 ports in a component with `'Sensor'` in the `'Name'` and run the query.

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

```
sensorComp =

  1×1 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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

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

Topics

"Create Architectural Views Programmatically"

Introduced in R2019b

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 object that the find method and the createView method use to select architecture elements with a stereotype that satisfies the given 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 =

  1×1 cell array

    {'KeylessEntryArchitecture/FOB 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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

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

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 has a given 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 if 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 by a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

`property` — Property
character vector

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

Data Types: `char`

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`getValue` | `setValue` | `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(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”. In order to import `requirementLinks`, you need a Simulink Requirements™ license.

`archModel = systemcomposer.importModel(importStruct)` creates a new architecture model based on a structure of MATLAB tables that specify components, ports, connections, port interfaces, and requirements.

`[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.

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 an 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. It 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.

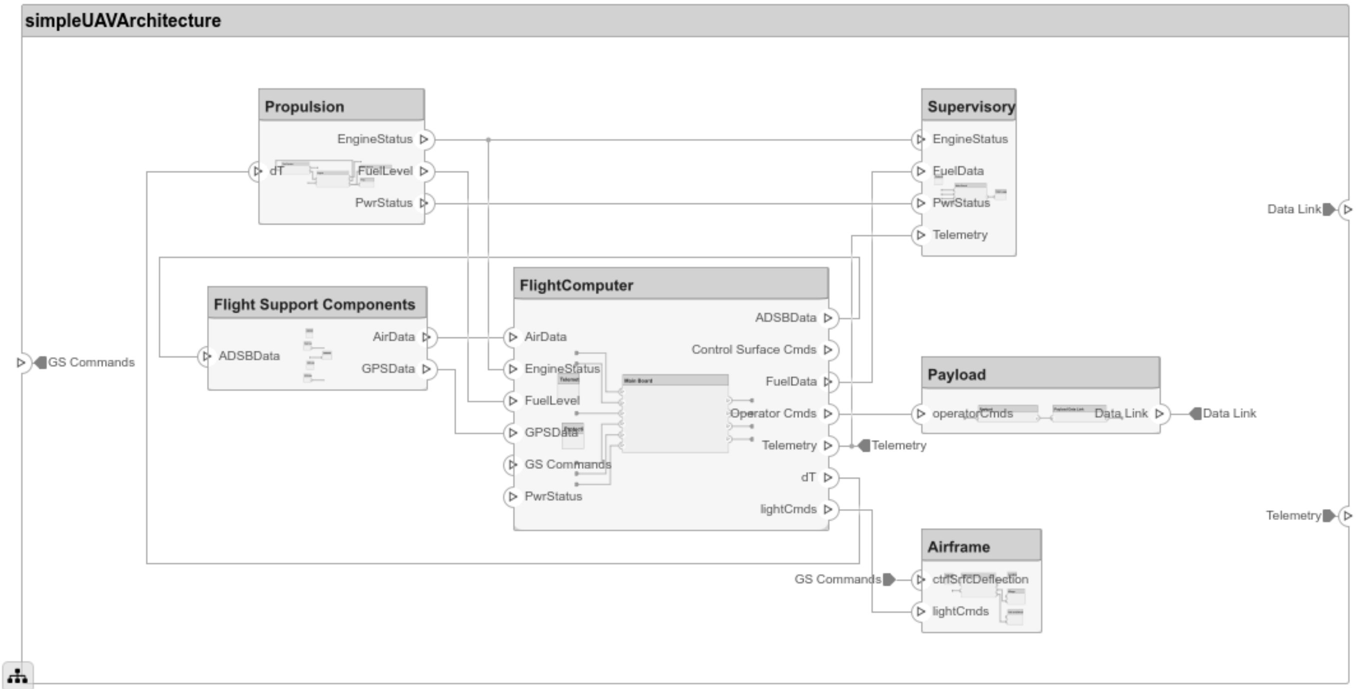
The systemcomposer.importModel(importModelName) API :

- 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 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);
```

Close Model

```
bdclose(modelName);
```

Input Arguments

`modelName` — Name of model

character vector

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

Example: `'importedModel'`

Data Types: `char`

`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 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 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.

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 Interfaces and Map Ports to Interfaces”. 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”. Optional columns include: DestinationArifact, 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 tables components, ports, connections, portInterfaces, and requirementLinks, Only the components table is required.

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
array of message objects

Errors reported during import process, returned as an array of message `MException` objects. You can obtain the error text by calling the `getString` method on each `MException` 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

Term	Definition	Application	More Information
requirements	A collection of statements describing the desired behavior and characteristics of a system. Requirements ensure system design integrity and are achievable, verifiable, unambiguous, and consistent with each other. Each level of design should have appropriate requirements.	To enhance traceability of requirements, link system, functional, customer, performance, or design requirements to components and ports. Link requirements to each other to represent derived or allocated requirements. Manage requirements from the requirements perspective on an architecture model or through custom views. Assign test cases to requirements.	• "Link and Trace Requirements" • "Manage Requirements" • "Update Reference Requirement Links from Imported File" on page 1-477

See Also

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

Topics

"Import and Export Architecture Models"

Introduced in R2019a

inlineComponent

Package: `systemcomposer.arch`

Inline reference architecture or behavior into model

Syntax

`componentObj = inlineComponent(component,inlineFlag)`

Description

`componentObj = inlineComponent(component,inlineFlag)` inlines the contents of the architecture model referenced by the specified `component`, and breaks the link to the reference model. If `inlineFlag` is 0 (`false`), then the contents are removed and only interfaces remain. You can also use `inlineComponent` to inline Stateflow Chart behaviors added to a component or to inline Simulink behaviors referenced by a component.

Examples

Reuse Component and Inline

Save the component `'robotComp'` in the architecture model `Robot.slx` and reference it from another component, `'electricComp'` so that `'electricComp'` uses the architecture of `'robotComp'`. Inline `'robotComp'` 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 `'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');
```

Inline `'robotComp'` 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 Inline

Add a Stateflow chart behavior to the component named `'robotComp'` within the current model. Inline 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));
```

Inline `'robotComp'` to remove the Stateflow Chart behavior. `inlineFlag` is ignored and set to `false`.

```
inlineComponent(comp(2));
```

Input Arguments

`component` — Architecture component
component object

Architecture component linked to an architecture model, specified as a `systemcomposer.arch.Component` object.

`inlineFlag` — Control of contents of inlined component
`true` or 1 | `false` or 0

Control of contents of inlined 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 inlined. If the component is a Simulink or Stateflow behavior, `inlineFlag` is ignored and set to `false`.

Data Types: `logical`

Output Arguments

`componentObj` — Architecture component
component object

Architecture component, 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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.	• “Implement Component Behavior in Simulink” • “Create a Reference Architecture”
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 an architectural component using state machines.	“Add Stateflow Chart Behavior to Architecture Component”
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.	• “Define Sequence Diagrams” • “Use Sequence Diagrams in the Views Gallery”

See Also

Reference Component | `isReference` | `linkToModel` | `saveAsModel`

Topics
"Decompose and Reuse Components"
"Add Stateflow Chart Behavior to Architecture Component"

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 of a model 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 of a model for analysis with all stereotypes in a profile.

`instance = instantiate( ___ ,Name,Value)` creates an instance of a model 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')
```

Analysis of Latency Characteristics

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

Create a 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('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','DefaultValue','true');
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','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('LatencyProfile'); % 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','LatencyProfile.NodeLatency');
```

```
batchApplyStereotype(arch,'Port','LatencyProfile.PortLatency');
batchApplyStereotype(arch,'Connector','LatencyProfile.ConnectorLatency');

% Instantiate using the analysis function
instance = instantiate(model.Architecture,'LatencyProfile','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('LatencyProfile.NodeLatency.resources')
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue('LatencyProfile.ConnectorLatency.secure')
```

```
defaultSecure = logical
   1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue('LatencyProfile.PortLatency.queueDep
```

```
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

Name of instance generated from the model, specified as a character vector.

Example: `'NewInstance'`

Data Types: `char`

`profile` — Profile name
character vector

Profile name, specified as a character vector.

Example: `'LatencyProfile'`

Data Types: `char`

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:
```
instantiate(model.Architecture,'LatencyProfile','NewInstance','Function',@cal
culateLatency,'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 the comma-separated pair consisting of `'NormalizeUnits'` and 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
MATLAB function handle

Analysis function, specified as the comma-separated pair consisting of `'Function'` and the MATLAB function handle to be executed when analysis is run.

`Arguments` — Analysis arguments
cell array of character vectors | character vector

Analysis arguments, specified as the comma-separated pair consisting of `'Arguments'` and a character vector or a cell array of character vectors of optional arguments to the analysis function.

Data Types: `char`

`Direction` — Analysis direction
`'TopDown'` | `'PreOrder'` | `'PostOrder'` | `'BottomUp'`

Analysis direction, specified as the comma-separated pair consisting of `'Direction'` and a character vector.

Data Types: `char`

`Strict` — Whether instances only get properties if the instance's specification has the stereotype applied
`false` or 0 (default) | `true` or 1

Whether instances only get properties if the instance's specification has the stereotype applied, specified as the comma-separated pair consisting of `'Strict'` and a logical 1 (`true`) or 0 (`false`).

Data Types: `logical`

Output Arguments

`instance` — Element instance
instance object

Element 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	“Analyze Architecture”
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	“Create a Model Instance for Analysis”

See Also

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

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 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 Whether Architecture Instance

Load the Small 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 by a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

Output Arguments

flag — Whether instance is architecture
`true` or 1 | `false` or 0

Whether instance is architecture, returned as a logical 1 (`true`) if the instance is an architecture or 0 (`false`) if the instance is not an architecture.

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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

`isComponent` | `isConnector` | `isPort` | `systemcomposer.analysis.Instance`

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 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 Whether Component Instance

Load the Small 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 by a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

Output Arguments

`flag` — Whether instance is component
`true` or 1 | `false` or 0

Whether instance is component, returned as a logical 1 (`true`) if the instance is a component or 0 (`false`) if the instance is not a component.

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 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 Whether Connector Instance

Load the Small 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 by a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

Output Arguments

`flag` — Whether instance is connector
`true` or 1 | `false` or 0

Whether instance is connector, returned as a logical 1 (`true`) if the instance is a connector or 0 (`false`) if the instance is not a connector.

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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` | `isPort` | `systemcomposer.analysis.Instance`

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` object that the `find` method and the `createView` method use to select a range of values from a specified `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 10ms to 40ms 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 name>.<stereotype name>.<property name>'` or any property on the designated class.

Example: `'Name'`

Example: `'AutoProfile.BaseComponent.Latency'`

Data Types: `char`

beginRangeValue — Beginning range value
value object

Beginning range value for `propertyName`, specified as a `systemcomposer.query.Value` object.

Example: `Value(20)`

Example: `Value(5,'ms')`

endRangeValue — Ending range value
value object

Ending range value for `propertyName`, specified as a `systemcomposer.query.Value` object.

Example: `Value(100)`

Example: `Value(20,'ms')`

Output Arguments

query — Query
query constraint object

Query, returned as a `systemcomposer.query.Constraint` object.

More About

Definitions

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

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 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 Whether Port Instance

Load the Small 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 by a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

`flag` — Whether instance is port
`true` or 1 | `false` or 0

Whether instance is port, returned as a logical 1 (`true`) if the instance is a port or 0 (`false`) if the instance is not a port.

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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

isReference

Package: `systemcomposer.arch`

Find if component is reference 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 to get port from
component object | variant component object

Component to get port from, 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 1 (`true`) if the component is a reference or 0 (`false`) if the component is not a reference.

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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.	• “Implement Component Behavior in Simulink” • “Create a Reference Architecture”

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 an architectural component using state machines.	“Add Stateflow Chart Behavior to Architecture Component”
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.	• “Define Sequence Diagrams” • “Use Sequence Diagrams in the Views Gallery”

See Also
Reference Component | `inlineComponent` | `linkToModel` | `saveAsModel`

Topics
“Implement Component Behavior in Simulink”
“Decompose and Reuse Components”

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` object that the `find` method and the `createView` method use to select a stereotype from the qualified `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 =

  1×1 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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• “Create Architecture Views Interactively” • “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.	“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 a Model Using Queries”

See Also

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

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 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 specified by one or more name-value pair arguments.

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

Examples

Battery Capacity Computation

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 to iterate over

architecture object | architecture instance object

Architecture to iterate over, 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

`iterFunction` — Iteration function
function handle

Iteration function, specified as a function handle to be iterated on each component.

Data Types: `string`

`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 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: `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 the comma-separated pair consisting of `'Recurse'` and 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 the comma-separated pair consisting of `'IncludePorts'` and 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 the comma-separated pair consisting of `'IncludeConnectors'` and 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 the comma-separated pair consisting of `'FollowConnectivity'` and 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"

Term	Definition	Application	More Information
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	“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.sldd');
linkDictionary(model,'newDictionary.sldd');
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

Dictionary file name with the `.sldd` extension, specified as a character vector. If a dictionary with this name does not exist, one will be created.

Example: `'dict_name.sldd'`

Data Types: `char`

More About

Definitions

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>architecture</td><td>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.</td><td>Different types of architectures describe different aspects of systems:
Functional architecture describes the flow of data in a system.
Logical architecture describes the intended operation of a system.
Physical architecture describes the platform or hardware in a system.</td><td>“Compose Architecture Visually”</td></tr>
<tr><td>model</td><td>A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.</td><td>Perform operations on a model:
Extract the root level architecture contained in the model.
Apply profiles.
Link interface data dictionaries.
Generate instances from model architecture.
System Composer models are stored as <code>.slx</code> files.</td><td>“Create an Architecture Model”</td></tr>
<tr><td>component</td><td>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.</td><td>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts.</td><td>“Components”</td></tr>
</table>

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"

Term	Definition	Application	More Information
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addReference` | `createDictionary` | `openDictionary` | `removeReference` | `saveToDictionary` | `unlinkDictionary`

Topics

"Save, Link, and Delete Interfaces"
"Reference 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 — Architecture component
component object

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

`modelName` — Model name
character vector

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

Example: `'Robot'`

Data Types: `char`

`modelFileName` — Model file name
character vector

Model file name for an existing model that defines the architecture or behavior of the component, specified as a character vector.

Example: `'Model.slx'`

Example: `'ProtectedModel.slxp'`

Data Types: `char`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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.	• "Implement Component Behavior in Simulink" • "Create a Reference Architecture"
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 an architectural component using state machines.	"Add Stateflow Chart Behavior to Architecture Component"
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.	• "Define Sequence Diagrams" • "Use Sequence Diagrams in the Views Gallery"

See Also

Reference Component | `inlineComponent` | `isReference` | `saveAsModel`

Topics

"Implement Component Behavior in Simulink"
"Decompose and Reuse Components"

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

Name of allocation set, specified as a character vector.

Example: `'MyNewAllocation'`

Data Types: `char`

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 a 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

`closeAll` | `createAllocationSet` | `open`

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: [1×5 systemcomposer.profile.Stereotype]
```

Input Arguments

`fileName` — File name for profile
character vector

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

Example: `'ProfileName.xml'`

Example: `'LatencyProfile'`

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

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

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

MAT-file that contains architecture instance, specified as a character vector.

Data Types: `char`

`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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"Create a Model Instance for Analysis"

See Also

`deleteInstance` | `instantiate` | `refresh` | `save` | `systemcomposer.analysis.Instance` | `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: [1×1 systemcomposer.arch.Architecture]
         SimulinkHandle: 2.0005
                  Views: [0×0 systemcomposer.view.ViewArchitecture]
               Profiles: [0×0 systemcomposer.profile.Profile]
    InterfaceDictionary: [1×1 systemcomposer.interface.Dictionary]
```

Input Arguments

`modelName` — Name of architecture model
character vector

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

Example: `'new_arch'`

Data Types: `char`

Output Arguments

`model` — Architecture model
model object

Architecture model, returned as a `systemcomposer.arch.Model` object.

More About

Definitions

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>architecture</td><td>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.</td><td>Different types of architectures describe different aspects of systems:
Functional architecture describes the flow of data in a system.
Logical architecture describes the intended operation of a system.
Physical architecture describes the platform or hardware in a system.</td><td>“Compose Architecture Visually”</td></tr>
<tr><td>model</td><td>A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.</td><td>Perform operations on a model:
Extract the root level architecture contained in the model.
Apply profiles.
Link interface data dictionaries.
Generate instances from model architecture.
System Composer models are stored as <code>.slx</code> files.</td><td>“Create an Architecture Model”</td></tr>
<tr><td>component</td><td>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.</td><td>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts.</td><td>“Components”</td></tr>
</table>

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 | save

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

Name of profile, specified as a character vector. Profile must be available on the MATLAB path with an `.xml` extension.

Example: `'new_profile'`

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

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

```
component = lookup(arch,'Path','RobotSystem/Sensors')
```

```
component = 

  Component with properties:

                 Name: 'Sensors'
               Parent: [1×1 systemcomposer.arch.Architecture]
                Ports: [1×2 systemcomposer.arch.ComponentPort]
           OwnedPorts: []
         Architecture: [1×1 systemcomposer.arch.Architecture]
    OwnedArchitecture: []
             Position: [275 75 391 161]
                Model: [1×1 systemcomposer.arch.Model]
                 UUID: 'f43c9d51-9dc6-43fc-b3af-95d458b81248'
       SimulinkHandle: 9.0002
  SimulinkModelHandle: 2.0002
          ExternalUID: ''
```

Input Arguments

`object` — Architecture model object
model object

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

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: `lookup(arch,'Path','RobotSystem/Sensors')`

`UUID` — Search by UUID
character vector

Search by UUID, specified as the comma-separated pair consisting of `'UUID'` and a character vector of the `UUID`.

Example: `lookup(arch,'UUID','f43c9d51-9dc6-43fc-b3af-95d458b81248')`

Data Types: `char`

`SimulinkHandle` — Search by simulink handle
double

Search by Simulink handle, specified as the comma-separated pair consisting of `'SimulinkHandle'` and a double of the `SimulinkHandle` value.

Example: `lookup(arch,'SimulinkHandle',9.0002)`

Data Types: `double`

`Path` — Search by full path
character vector

Search by file path, specified as the comma-separated pair consisting of `'Path'` and a character vector with the path defined.

Example: `lookup(arch,'Path','RobotSystem/Sensors')`

Data Types: `char`

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`, or `systemcomposer.arch.Connector` 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"Create a Model Instance for Analysis"

See Also

`createView` | `find` | `instantiate` | `iterate` | `systemcomposer.analysis.Instance` | `systemcomposer.view.ElementGroup` | `systemcomposer.view.View`

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 `component` to a variant choice component and returns the parent variant component and available variant choice components.

`[variantComp,choices] = makeVariant(component,Name,Value)` converts `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: [1×1 systemcomposer.arch.Architecture]
                   Name: 'Component'
                 Parent: [1×1 systemcomposer.arch.Architecture]
```

```
                 Ports: [1×2 systemcomposer.arch.ComponentPort]
            OwnedPorts: [1×2 systemcomposer.arch.ComponentPort]
     OwnedArchitecture: [1×1 systemcomposer.arch.Architecture]
              Position: [15 13 65 81]
                 Model: [1×1 systemcomposer.arch.Model]
        SimulinkHandle: 69.0001
   SimulinkModelHandle: 1.2207e-04
                  UUID: 'ee705b8f-b383-4230-a1a2-3c69fb081cc5'
           ExternalUID: ''


choices = 

  Component with properties:

     IsAdapterComponent: 0
           Architecture: [1×1 systemcomposer.arch.Architecture]
                   Name: 'Component'
                 Parent: [1×1 systemcomposer.arch.Architecture]
                  Ports: [1×2 systemcomposer.arch.ComponentPort]
             OwnedPorts: [1×2 systemcomposer.arch.ComponentPort]
      OwnedArchitecture: [1×1 systemcomposer.arch.Architecture]
               Position: [50 20 100 76]
                  Model: [1×1 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 a variant choice component, specified as a `systemcomposer.arch.Component` object.

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: `[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

Name of variant component, specified as the comma-separated pair consisting of `'Name'` and a character vector.

Example: `[variantComp,choices] = makeVariant(newComponent,'Name','NewVariantComponent')`

`Label` — Label of variant choice
character vector

Label of variant choice from converted component, specified as the comma-separated pair consisting of `'Label'` and a character vector.

Example: `[variantComp,choices] = makeVariant(newComponent,'Name','NewVariantComponent','Label','NewVariantChoice')`

`Choices` — Variant choice names
cell array of character vectors

Variant choice names, specified as the comma-separated pair consisting of `'Choices'` and a cell array of character vectors. The 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'})`

`ChoiceLabels` — Variant choice labels
cell array of character vectors

Variant choice labels, specified as the comma-separated pair consisting of `'ChoiceLabels'` and a cell array of character vectors.

Example: `[variantComp,choices] = makeVariant(newComponent,'Choices',{'NewVariantChoiceA','NewVariantChoiceB','NewVariantChoiceC'},'ChoiceLabels',{'Choice A','Choice B','Choice C'})`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 1-417

See Also

Variant Component | `addChoice` | `addVariantComponent` | `getChoices`

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

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

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

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

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 in the allocation editor if the allocation set is on the MATLAB path.

Examples

Create Allocation Set and Open

```
% 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

Name of allocation set, specified as an `systemcomposer.allocation.AllocationSet` object or the name as a character vector.

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 a 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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"

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>port</td><td>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.</td><td>There are different types of ports:
Component ports are interaction points on the component to other components.
Architecture ports are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</td><td>“Ports”</td></tr>
<tr><td>connector</td><td>Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.</td><td>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.</td><td>“Connections”</td></tr>
</table>

See Also

`createModel` | `openModel`

Topics

“Create an Architecture Model”

Introduced in R2019a

systemcomposer.openDictionary

Open data dictionary

Syntax

`dict_id = systemcomposer.openDictionary(dictionaryName)`

Description

`dict_id = systemcomposer.openDictionary(dictionaryName)` opens an existing Simulink data dictionary to hold interfaces and returns the `systemcomposer.interface.Dictionary` object.

Examples

Open Existing Dictionary

Create a dictionary and open the dictionary.

```
systemcomposer.createDictionary('my_dictionary.sldd');
dict_id = systemcomposer.openDictionary('my_dictionary.sldd');
```

Input Arguments

`dictionaryName` — Name of existing data dictionary
character vector

Name of existing data dictionary, specified as a character vector. The name must include the `.sldd` extension.

Example: `'my_dictionary.sldd'`

Data Types: `char`

Output Arguments

`dict_id` — 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"

Term	Definition	Application	More Information
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addReference` | `createDictionary` | `linkDictionary` | `removeReference` | `saveToDictionary` | `unlinkDictionary`

Topics

"Save, Link, and Delete Interfaces"
"Reference 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: [1×1 systemcomposer.arch.Architecture]
       SimulinkHandle: 2.0005
                Views: [0×0 systemcomposer.view.ViewArchitecture]
             Profiles: [0×0 systemcomposer.profile.Profile]
  InterfaceDictionary: [1×1 systemcomposer.interface.Dictionary]
```

Input Arguments

`modelName` — Name of model
character vector

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

Example: `'new_arch'`

Data Types: `char`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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

`close` | `open`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries”

See Also

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

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` object that the `find` method and the `createView` method use to select non-evaluated values for object properties or stereotype properties for elements based on a specified property `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 =

1×1 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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

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

Topics
"Create Architectural Views Programmatically"

Introduced in R2019b

PropertyValue

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` object that the `find` method and the `createView` method use to select object properties or stereotype properties for elements based on specified property `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 =

4×1 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'   }
```

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

Property | createView | find | systemcomposer.query.Constraint

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 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 the 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 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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	“Analyze Architecture”
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	“Create a Model Instance for Analysis”

See Also

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

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 keeping the contents, and then 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 editor to see it.

```
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 editor.

```
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 the component
character vector

Path to the component including the name of the top-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 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

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

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

Introduced in R2019b

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'`

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

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

Topics
"Create Architecture Views Interactively"
"Create Architectural Views Programmatically"

Introduced in R2021a

removeElement

Package: `systemcomposer.interface`

Remove signal interface element

Syntax

`removeElement(interface,elementName)`

Description

`removeElement(interface,elementName)` removes an element from a signal interface.

Examples

Add Interface and Element then Remove Element

Add an interface `'newInterface'` to the interface dictionary of the model and add an element with type `'double'` to it, then remove the element.

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

Input Arguments

`interface` — Interface
signal interface object

Interface, specified as a `systemcomposer.interface.SignalInterface` object.

`elementName` — Name of element
character vector

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

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>adapter</td><td>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.</td><td>With an adapter, you can perform three functions on the Interface Adapter dialog:
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.</td><td>“Interface Adapter”</td></tr>
</table>

See Also

Adapter | `addElement` | `getDestinationElement` | `getElement` | `getSourceElement`

Topics

“Define Interfaces”

Introduced in R2019a

removeInterface

Package: systemcomposer.interface

Remove named interface from interface dictionary

Syntax

removeInterface(dictionary,name)

Description

removeInterface(dictionary,name) removes a named interface from the interface dictionary.

Examples

Remove Interface

Add an interface 'newInterface' to the interface dictionary of the model and then remove it.

Create a new model, and add an interface to the interface dictionary of the model.

```
arch = systemcomposer.createModel('archModel');
addInterface(arch.InterfaceDictionary,'newInterface');
```

Open the model, and open the interface editor. Confirm an interface named 'newInterface' exists.

```
open(arch)
```

Remove the interface.

```
removeInterface(arch.InterfaceDictionary,'newInterface');
```

View the interface editor. Confirm an interface named '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

Name of interface to be removed, specified as a character vector.

Data Types: char

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	“Assign Interfaces to Ports”
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• “Save, Link, and Delete Interfaces” • “Reference Data Dictionaries”

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

Adapter | `addInterface` | `getInterface` | `getInterfaceNames`

Topics

"Define Interfaces"

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

Name of profile, specified as a character vector.

Example: `'SystemProfile'`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

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

Name of property to be removed, specified as a character vector.

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

`addProperty` | `getProperty` | `setProperty`

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• “Create Architecture Views Interactively” • “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.	“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 a Model Using Queries”

See Also

`createView` | `deleteView` | `getView` | `modifyQuery` | `openViews` | `runQuery` | `systemcomposer.view.ElementGroup` | `systemcomposer.view.View`

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

Referenced dictionary, specified as a character vector of the name of the referenced dictionary with the `.sldd` extension.

Example: `'ReferenceDictionary.sldd'`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addReference` | `createDictionary` | `linkDictionary` | `openDictionary` | `saveToDictionary` | `unlinkDictionary`

Topics

"Save, Link, and Delete Interfaces"
"Reference 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 | stereotype object

Stereotype to remove, specified as a character vector or a systemcomposer.profile.Stereotype object.

Example: 'electricalComponent'

Data Types: char

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`addStereotype` | `getDefaultStereotype` | `getStereotype` | `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 | signal interface object

Model element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, `systemcomposer.arch.Connector`, or `systemcomposer.interface.SignalInterface` object.

`stereotype` — Stereotype
character vector

Stereotype, specified as a character vector in the form `'<profile>.<stereotype>'`. The profile must already be applied to the model.

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

`applyStereotype` | `batchApplyStereotype` | `getStereotypes`

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

Model architecture, specified as a `systemcomposer.arch.Model` object or a character vector as the name of the model.

Example: `'MyModel'`

Example: `archModel`

Data Types: `char`

`oldProfileName` — Old profile name
character vector

Old profile name, specified as a character vector.

Example: `'MyProfile'`

Data Types: `char`

`newProfileName` — New profile name
character vector

New profile name, specified as a character vector.

Example: `'MyProfileNew'`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "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.	"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 a Model Using Queries"

See Also

`createView` | `deleteView` | `getView` | `modifyQuery` | `openViews` | `removeQuery` | `systemcomposer.view.ElementGroup` | `systemcomposer.view.View`

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

Path to save, specified as a character vector. The current directory is the default if no path is specified.

Example: `'C:\Temp\MATLAB'`

Data Types: `char`

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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`close` | `closeAll` | `createProfile` | `editor` | `find` | `load` | `open`

Topics

"Define Profiles and Stereotypes"

Introduced in R2019a

save

Package: `systemcomposer.allocation`

Save allocation set

Syntax

`save(allocSet)`

Description

`save(allocSet)` saves the allocation set.

Examples

Create Allocation Set and Save

```
% 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.

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 a 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` | `systemcomposer.allocation.AllocationSet`

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

```
save(arch);
save(arch.InterfaceDictionary);
```

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"

Term	Definition	Application	More Information
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`close` | `loadModel`

Topics

"Create an Architecture Model"
"Save, Link, and Delete Interfaces"

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

`MAT`-file to save instance, specified as a character vector.

Example: `'InstanceFile'`

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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"Create a Model Instance for Analysis"

See Also

`deleteInstance` | `instantiate` | `iterate` | `loadInstance` | `lookup` | `refresh` | `systemcomposer.analysis.Instance` | `update`

Topics

"Write Analysis Function"

Introduced in R2019a

saveAsModel

Package: systemcomposer.arch

Save architecture of component to separate model

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.

Examples

Save Component

Save the 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 to the model with the names 'electricComp' and 'robotComp'.

```
names = {'electricComp','robotComp'};
comp = addComponent(arch,names);
```

Save the 'robotComp' component in a model so the component references the architecture model Robot.slx.

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

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

Model name, specified 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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.	• "Implement Component Behavior in Simulink" • "Create a Reference Architecture"
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 an architectural component using state machines.	"Add Stateflow Chart Behavior to Architecture Component"
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.	• "Define Sequence Diagrams" • "Use Sequence Diagrams in the Views Gallery"

See Also

Reference Component | `inlineComponent` | `isReference` | `linkToModel`

Topics
"Implement Component Behavior in Simulink"
"Decompose and Reuse Components"

Introduced in R2019a

saveToDictionary

Package: `systemcomposer.arch`

Save interfaces to dictionary

Syntax

```
saveToDictionary(model,dictionaryName)
saveToDictionary(model,dictionaryName,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(model,dictionaryName,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.

`dictionaryName` — Dictionary name
character vector

Dictionary name, specified as a character vector. If a dictionary with this name does not exist, one will be created.

Example: `'dict_name'`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addReference` | `createDictionary` | `linkDictionary` | `openDictionary` | `removeReference` | `unlinkDictionary`

Topics

"Save, Link, and Delete Interfaces"
"Reference 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 | label of variant choice

Active choice in a variant component, specified as a `systemcomposer.arch.Component` object or label of the variant choice as a character vector.

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 1-417

See Also

Variant Component | `addChoice` | `addVariantComponent` | `getActiveChoice` | `getChoices`

Topics

"Create Variants"

Introduced in R2019a

setComplexity

Package: `systemcomposer.interface`

Set complexity for signal interface element

Syntax

`setComplexity(interfaceElem,complexity)`

Description

`setComplexity(interfaceElem,complexity)` sets the complexity for the designated signal interface element.

Examples

Set Complexity for Interface Element

Set the complexity for an interface element.

Create a model named `'archModel'`.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add an interface, then create an interface element with the name `'x'`.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
elem = interface.addElement('x'); % Create interface element
```

Set the complexity for the interface element as `'complex'`.

```
setComplexity(elem,'complex'); % Set complexity for interface element
```

Input Arguments

`interfaceElem` — Interface element

signal element object

Interface element, specified as a `systemcomposer.interface.SignalElement` object.

`complexity` — Complexity of interface element

`'real'` (default) | `'complex'`

Complexity of interface element, specified as `'real'` or `'complex'`.

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	“Assign Interfaces to Ports”
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• “Save, Link, and Delete Interfaces” • “Reference Data Dictionaries”

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addElement` | `addInterface` | `createModel` | `systemcomposer.interface.SignalElement`

Topics

"Define Interfaces"

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

Control string that controls the selection of choice, specified 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 1-417

See Also

Variant Component | `addChoice` | `addVariantComponent` | `getActiveChoice` | `getCondition` | `makeVariant` | `setActiveChoice`

Topics

"Create Variants"

Introduced in R2019a

setDefaultComponentStereotype

Package: `systemcomposer.profile`

Set default stereotype for components

Syntax

`setDefaultComponentStereotype(stereotype,stereotypeName)`

Description

`setDefaultComponentStereotype(stereotype,stereotypeName)` specifies the default stereotype `stereotypeName` of the children whose parent component has `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.setDefaultComponentStereotype('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. 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 stereotypes on the child component.

```
childComponent = addComponent(newComponent.Architecture,'Child');
```

```
stereotypes = getStereotypes(childComponent)
```

```
stereotypes =

  1×1 cell array

    {'LatencyProfile.NodeLatency'}
```

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

Default stereotype name for child components, specified as a character vector 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`applyStereotype` | `removeStereotype` | `setDefaultConnectorStereotype` | `setDefaultPortStereotype`

Topics

"Define Profiles and Stereotypes"

Introduced in R2019a

setDefaultConnectorStereotype

Package: `systemcomposer.profile`

Set default stereotype for connectors

Syntax

`setDefaultConnectorStereotype(stereotype,stereotypeName)`

Description

`setDefaultConnectorStereotype(stereotype,stereotypeName)` specifies the default stereotype `stereotypeName` of the connectors within the parent component that has `stereotype` applied.

Examples

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.setDefaultConnectorStereotype('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. 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, ports, a connection between them, and get stereotypes on the connector.

```
childComponent1 = addComponent(newComponent.Architecture,'Child1');
childComponent2 = addComponent(newComponent.Architecture,'Child2');

outPort1 = addPort(childComponent1.Architecture,'testSig','out');
inPort1 = addPort(childComponent2.Architecture,'testSig','in');

srcPort = getPort(childComponent1,'testSig');
destPort = getPort(childComponent2,'testSig');

connector = connect(srcPort,destPort);

stereotypes = getStereotypes(connector)
```

```
stereotypes =

  1×1 cell array

    {'LatencyProfile.ConnectorLatency'}
```

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

Default stereotype name for connectors, specified as a character vector 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

`applyStereotype` | `removeStereotype` | `setDefaultComponentStereotype` | `setDefaultPortStereotype`

Topics

“Define Profiles and Stereotypes”

Introduced in R2019a

setDefaultPortStereotype

Package: `systemcomposer.profile`

Set default stereotype for ports

Syntax

`setDefaultPortStereotype(stereotype,stereotypeName)`

Description

`setDefaultPortStereotype(stereotype,stereotypeName)` specifies the default stereotype `stereotypeName` of the ports on the architecture of the parent component that has `stereotype` applied.

Examples

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.setDefaultPortStereotype('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. 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 stereotypes.

```
port = addPort(newComponent.Architecture,'testSig','out');
```

```
stereotypes = getStereotypes(port)
```

```
stereotypes =

  1×1 cell array

    {'LatencyProfile.PortLatency'}
```

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

Default stereotype name for ports, specified as a character vector 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`applyStereotype` | `removeStereotype` | `setDefaultComponentStereotype` | `setDefaultConnectorStereotype`

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 =

  1×1 cell array
```

```
{'LatencyProfile.NodeLatency'}
```

Input Arguments

`profile` — Profile
profile object

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

`stereotypeName` — Stereotype name
character vector

Stereotype name, specified as a character vector. The stereotype must be present in the profile.

Example: `'ComponentStereotype'`

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`addStereotype` | `createProfile` | `getDefaultStereotype` | `getStereotype` | `removeStereotype`

Topics

"Create a Profile and Add Stereotypes"

Introduced in R2019a

setDescription

Package: `systemcomposer.interface`

Set description for signal interface element

Syntax

`setDescription(interfaceElem,description)`

Description

`setDescription(interfaceElem,description)` sets the description for the designated signal interface element.

Examples

Set Description for Interface Element

Set the description for an interface element.

Create a model named `'archModel'`.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add an interface, then create an interface element with the name `'x'`.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
elem = interface.addElement('x'); % Create interface element
```

Set the description for the interface element as `'Test Description'`.

```
setDescription(elem,'Test Description'); % Set description for interface element
```

Input Arguments

`interfaceElem` — Interface element
signal element object

Interface element, specified as a `systemcomposer.interface.SignalElement` object.

`description` — Description of interface element
character vector

Description of interface element, specified as a character vector.

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	“Assign Interfaces to Ports”
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• “Save, Link, and Delete Interfaces” • “Reference Data Dictionaries”

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addElement` | `addInterface` | `createModel` | `systemcomposer.interface.SignalElement`

Topics

"Define Interfaces"

Introduced in R2019a

setDimensions

Package: `systemcomposer.interface`

Set dimensions for signal interface element

Syntax

`setDimensions(interfaceElem,dimensions)`

Description

`setDimensions(interfaceElem,dimensions)` sets the dimensions for the designated signal interface element.

Examples

Set Dimensions for Interface Element

Set the dimensions for an interface element.

Create a model named `'archModel'`.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add an interface, then create an interface element with the name `'x'`.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
elem = interface.addElement('x'); % Create interface element
```

Set the dimensions for the interface element as `'2'`.

```
setDimensions(elem,'2'); % Set dimensions for interface element
```

Input Arguments

`interfaceElem` — Interface element
signal element object

Interface element, specified as a `systemcomposer.interface.SignalElement` object.

`dimensions` — Dimensions of interface element
character vector

Dimensions of interface element, specified as a character vector.

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addElement` | `addInterface` | `createModel` | `systemcomposer.interface.SignalElement`

Topics

"Define Interfaces"

Introduced in R2019a

setMaximum

Package: `systemcomposer.interface`

Set maximum for signal interface element

Syntax

`setMaximum(interfaceElem,maximum)`

Description

`setMaximum(interfaceElem,maximum)` sets the maximum for the designated signal interface element.

Examples

Set Maximum for Interface Element

Set the maximum value for an interface element.

Create a model named `'archModel'`.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add an interface, then create an interface element with the name `'x'`.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
elem = interface.addElement('x'); % Create interface element
```

Set the maximum for the interface element as `'5.72'`.

```
setMaximum(elem,'5.72'); % Set maximum for interface element
```

Input Arguments

`interfaceElem` — Interface element

signal element object

Interface element, specified as a `systemcomposer.interface.SignalElement` object.

`maximum` — Maximum of interface element

character vector

Maximum of interface element, specified as a character vector.

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addElement` | `addInterface` | `createModel` | `systemcomposer.interface.SignalElement`

Topics

"Define Interfaces"

Introduced in R2019a

setMinimum

Package: systemcomposer.interface

Set minimum for signal interface element

Syntax

```
setMinimum(interfaceElem,minimum)
```

Description

`setMinimum(interfaceElem,minimum)` sets the minimum for the designated signal interface element.

Examples

Set Minimum for Interface Element

Set the minimum value for an interface element.

Create a model named `'archModel'`.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add an interface, then create an interface element with the name `'x'`.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
elem = interface.addElement('x'); % Create interface element
```

Set the minimum for the interface element as `'1.12'`.

```
setMinimum(elem,'1.12'); % Set minimum for interface element
```

Input Arguments

`interfaceElem` — Interface element

signal element object

Interface element, specified as a `systemcomposer.interface.SignalElement` object.

`minimum` — Minimum of interface element

character vector

Minimum of interface element, specified as a character vector.

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addElement` | `addInterface` | `createModel` | `systemcomposer.interface.SignalElement`

Topics

"Define Interfaces"

Introduced in R2019a

setName

Package: `systemcomposer.interface`

Set name for signal interface element

Syntax

`setName(interfaceElem,name)`

Description

`setName(interfaceElem,name)` sets the name for the designated signal interface element.

Examples

Set Name for Interface Element

Set the name for an interface element.

Create a model named `'archModel'`.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add an interface, then create an interface element with the name `'x'`.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
elem = interface.addElement('x'); % Create interface element
```

set a new name for the interface element as `'newName'`.

```
setName(elem,'newName'); % Set new name for interface element
```

Input Arguments

`interfaceElem` — Interface element
signal element object

Interface element to be named, specified as a `systemcomposer.interface.SignalElement` object.

name — Name of interface element
character vector

Name of interface element, specified as a character vector.

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addElement` | `addInterface` | `createModel` | `systemcomposer.interface.SignalElement`

Topics

"Define Interfaces"

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

Name of port, specified 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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

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

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 an interface and set the interface for the port.

```
newInterface = addInterface(model.InterfaceDictionary,'NewInterface');
setInterface(newPort,newInterface);
```

Input Arguments

port — Port to be edited
port object

Port to be edited, specified as a systemcomposer.arch.ArchitecturePort or systemcomposer.arch.ComponentPort object.

interface — Interface
signal interface object

Interface to set, specified as a systemcomposer.interface.SignalInterface 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"

Term	Definition	Application	More Information
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

Component | `systemcomposer.arch.ArchitecturePort` | `systemcomposer.arch.ComponentPort`

Topics

"Define Interfaces"

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)
```

Input Arguments

element — Architecture model element

architecture object | component object | port object | connector object

Architecture model element, specified as a `systemcomposer.arch.Architecture`, `systemcomposer.arch.Component`, `systemcomposer.arch.VariantComponent`, `systemcomposer.arch.ComponentPort`, `systemcomposer.arch.ArchitecturePort`, or `systemcomposer.arch.Connector` object.

propertyName — Name of property

character vector

Name of property, specified as a character vector in the form `'<profile>.<stereotype>.<property>'`.

Data Types: `char`

propertyValue — Value of property

character vector

Value of property, specified as a character vector. Specify numeric values in single quotes. 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 1-457.

Data Types: `char`

propertyUnits — Units of property

character vector

Units of property to interpret property values, specified as a character vector.

Data Types: `char`

More About

Definitions

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>architecture</td><td>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.</td><td>Different types of architectures describe different aspects of systems:
Functional architecture describes the flow of data in a system.
Logical architecture describes the intended operation of a system.
Physical architecture describes the platform or hardware in a system.</td><td>“Compose Architecture Visually”</td></tr>
<tr><td>model</td><td>A System Composer model is the file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.</td><td>Perform operations on a model:
Extract the root level architecture contained in the model.
Apply profiles.
Link interface data dictionaries.
Generate instances from model architecture.
System Composer models are stored as <code>.slx</code> files.</td><td>“Create an Architecture Model”</td></tr>
<tr><td>component</td><td>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.</td><td>Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts.</td><td>“Components”</td></tr>
</table>

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	“Use Stereotypes and Profiles”
property	A property is a field in a stereotype. For each model 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.	“Set Properties”

See Also

addProperty | getProperty | removeProperty

Topics

"Set Properties for Analysis"

Introduced in R2019a

setType

Package: `systemcomposer.interface`

Set type for signal interface element

Syntax

`setType(interfaceElem,type)`

Description

`setType(interfaceElem,type)` sets the type for the designated signal interface element.

Examples

Set Type for Interface Element

Set the type for an interface element.

Create a model named `'archModel'`.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add an interface, then create an interface element with the name `'x'`.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
elem = interface.addElement('x'); % Create interface element
```

Set the type for the interface element as `'single'`.

```
setType(elem,'single'); % Set type for interface element
```

Input Arguments

`interfaceElem` — Interface element
signal element object

Interface element, specified as a `systemcomposer.interface.SignalElement` object.

`type` — Type of interface element
character vector

Type of interface element, specified as a character vector for a valid MATLAB data type.

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addElement` | `addInterface` | `createModel` | `systemcomposer.interface.SignalElement`

Topics

"Define Interfaces"

Introduced in R2019a

setUnits

Package: `systemcomposer.interface`

Set units for signal interface element

Syntax

`setUnits(interfaceElem,units)`

Description

`setUnits(interfaceElem,units)` sets the units for the designated signal interface element.

Examples

Set Units for Interface Element

Set the units for an interface element.

Create a model named `'archModel'`.

```
modelName = 'archModel';
arch = systemcomposer.createModel(modelName,true); % Create model
```

Add an interface, then create an interface element with the name `'x'`.

```
interface = arch.InterfaceDictionary.addInterface('interface'); % Add interface
elem = interface.addElement('x'); % Create interface element
```

Set the units for the interface element as `'kg'`.

```
setUnits(elem,'kg'); % Set units for interface element
```

Input Arguments

`interfaceElem` — Interface element
signal element object

Interface element, specified as a `systemcomposer.interface.SignalElement` object.

`units` — Units of interface element
character vector

Units of interface element, specified as a character vector.

Data Types: `char`

More About

Definitions

Term	Definition	Application	More Information
interface	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	“Interface Adapter”

See Also

`addElement` | `addInterface` | `createModel` | `systemcomposer.interface.SignalElement`

Topics

“Define Interfaces”

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` of the `instance` to `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 by a `systemcomposer.analysis.ArchitectureInstance`, `systemcomposer.analysis.ComponentInstance`, `systemcomposer.analysis.PortInstance`, or `systemcomposer.analysis.ConnectorInstance` object.

`property` — Property
character vector

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

`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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`getValue` | `hasValue` | `systemcomposer.analysis.Instance`

Topics

"Write Analysis Function"

Introduced in R2019a

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"

Term	Definition	Application	More Information
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addReference` | `createDictionary` | `linkDictionary` | `removeReference` | `saveToDictionary`

Topics

"Save, Link, and Delete Interfaces"
"Reference 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. 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 to be 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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"Create a Model Instance for Analysis"

See Also

`deleteInstance` | `instantiate` | `iterate` | `loadInstance` | `lookup` | `refresh` | `save` | `systemcomposer.analysis.Instance`

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.

Examples

Update Reference Requirement Links from Imported File

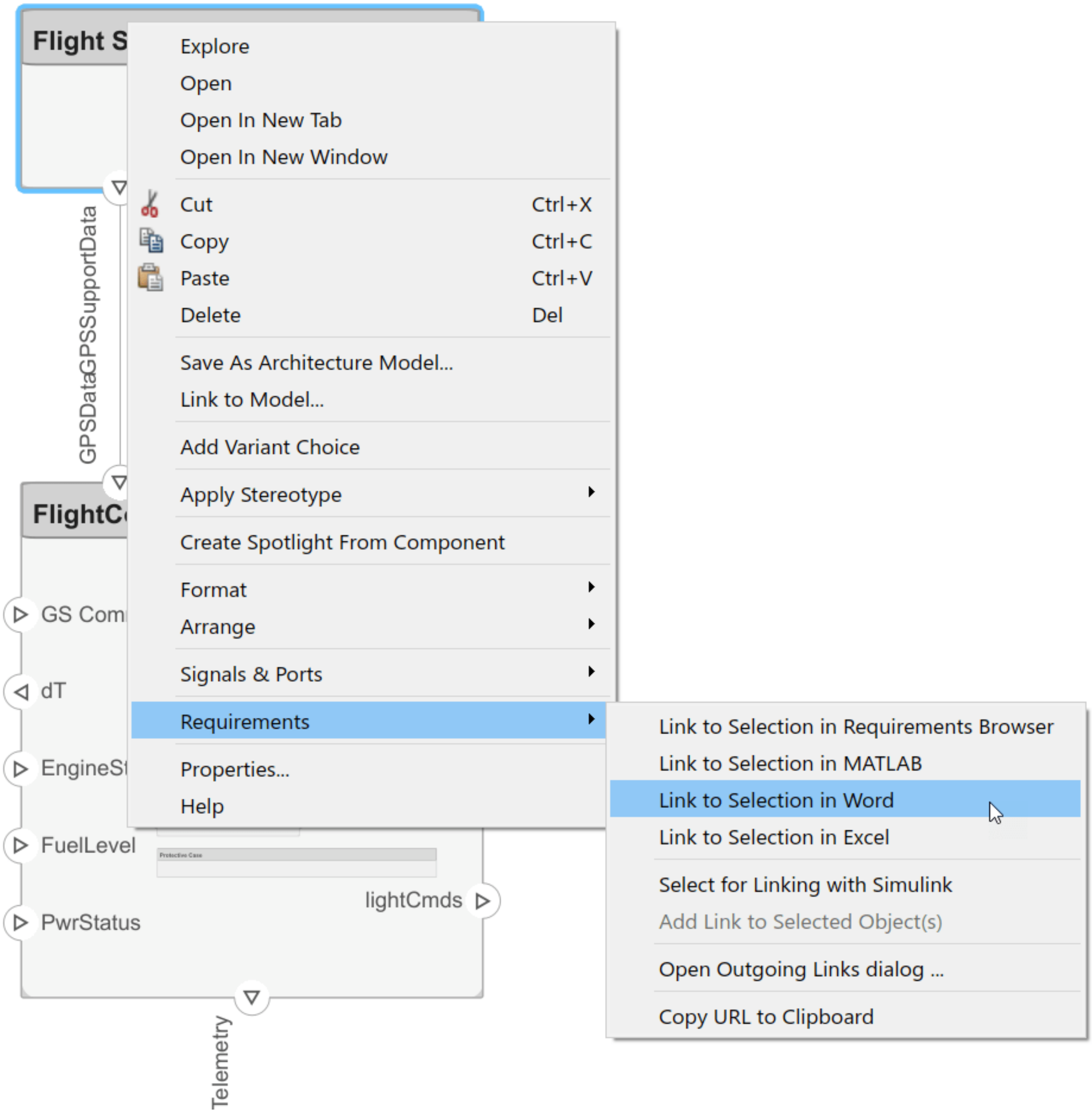
After importing requirement links from a file, update links to reference requirements for the model to make full use of the Simulink® Requirements™ functionality.

```
model = systemcomposer.openModel('reqImportExample');
```

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. From the drop-down list, select **Requirements > Link Selection to Word**.



Export Model and Save to External Files

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 files to import requirement links into another model.

```
structModel = ImportModelFromExcel('exportedModel.xls','Components','Ports', ...
'Connections','PortInterfaces','RequirementLinks');
structModel.readTableFromExcel;

arch = 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.

```
close(model);
model2 = systemcomposer.openModel('reqNewExample');
systemcomposer.updateLinksToReferenceRequirements('reqNewExample','linktype_rmi_word','Functiona
```

Input Arguments

modelName — Name of model
character vector

Name of model, specified as a character vector.

Data Types: char

linkDomain — Link domain
character vector

Link domain, specified as a character vector. 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

documentPathOrID — Full document path
character vector

Full document path, specified as a character vector.

Example: 'Functional_Requirements.docx'

Data Types: char

More About

Definitions

Term	Definition	Application	More Information
requirements	A collection of statements describing the desired behavior and characteristics of a system. Requirements ensure system design integrity and are achievable, verifiable, unambiguous, and consistent with each other. Each level of design should have appropriate requirements.	To enhance traceability of requirements, link system, functional, customer, performance, or design requirements to components and ports. Link requirements to each other to represent derived or allocated requirements. Manage requirements from the requirements perspective on an architecture model or through custom views. Assign test cases to requirements.	• “Link and Trace Requirements” • “Manage Requirements” • “Update Reference Requirement Links from Imported File” on page 1-477

See Also

`exportModel` | `importModel`

Topics

“Link and Trace Requirements”
“Manage Requirements”
“Import and Export Architecture Models”
“Custom Link Types” (Simulink Requirements)

Introduced in R2020b

2

Classes

systemcomposer.allocation.Allocation

Class that represents allocation between source and target element

Description

The `systemcomposer.allocation.Allocation` defines the allocation between the source element and the target element.

Related classes 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`

Examples

Allocate Architectures in a Tire Pressure Monitoring System

This example shows how to use allocations to analyze a tire pressure monitoring system.

Overview

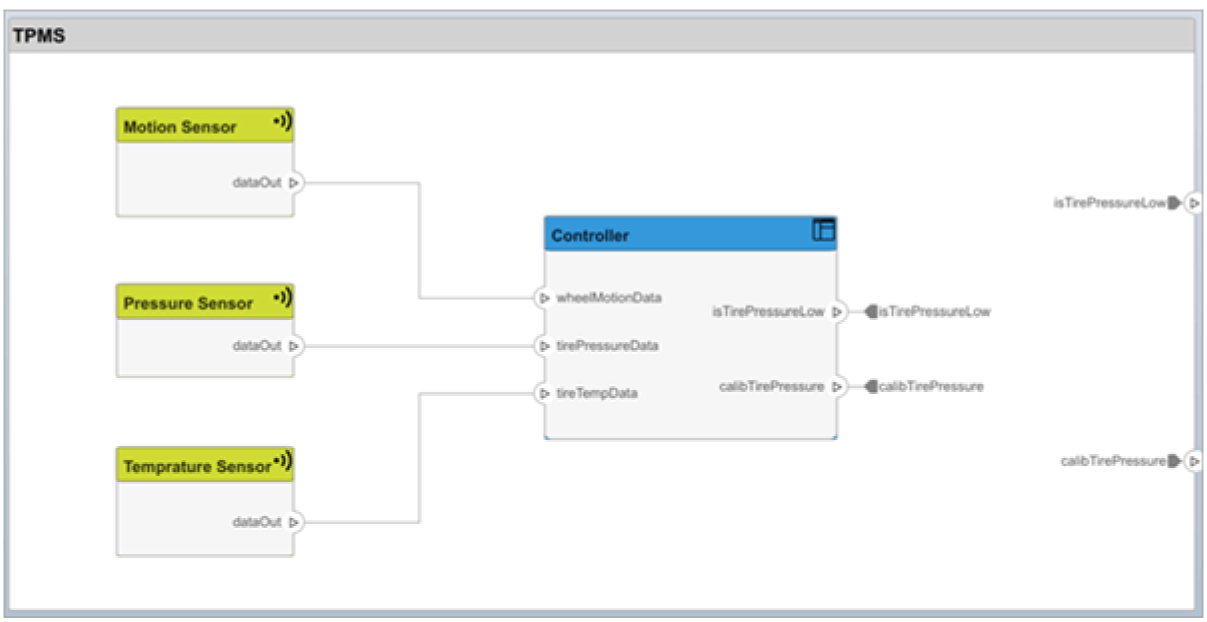
In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system 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.

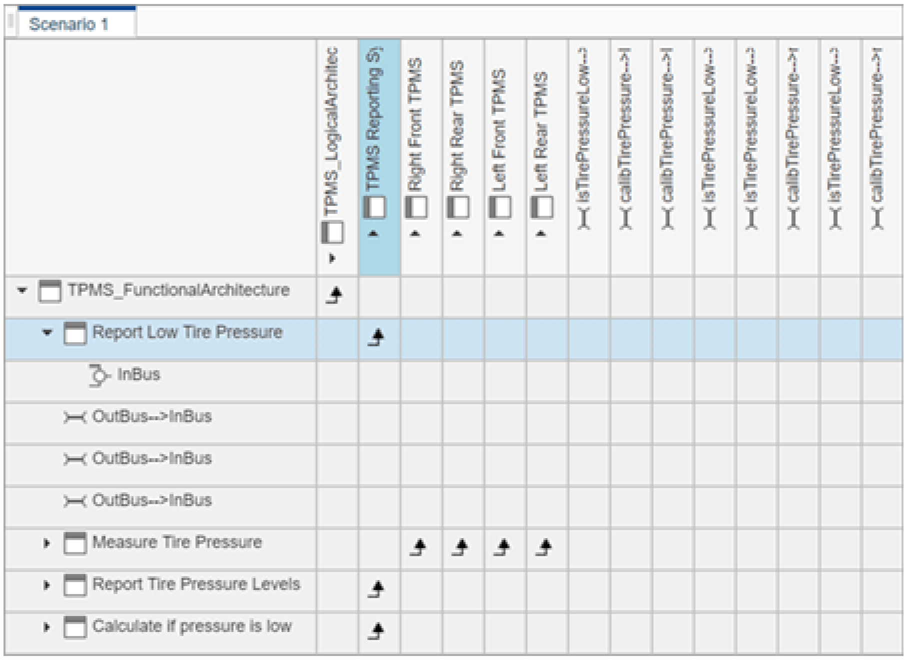
Use this command to open the project.

```
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 and the elements of

`TPMS_LogicalArchitecture` are displayed in the first row. The arrows indicate the allocations between model elements.



This figure displays allocations in the architectural component level. 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 you can 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 example shows how to identify which functions will be provided by which suppliers using the specified allocations. The supplier information is stored in the logical model, since these are the components that the suppliers will be delivering to the system integrator.

```
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, nu
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.Supplie
        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
Report Low Tire Pressure	1	0	0	0
Measure temprature of tire	0	0	0	1
Calculate Tire Pressure	0	1	0	0
Measure rotations	0	1	0	0
Calculate if pressure is low	1	0	0	0
Report Tire Pressure Levels	1	0	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 of the components in the ECU, accumulate the binary size required for each of the allocated software components.

```
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.Bi
        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 a 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` | `getAllocatedFrom` | `getAllocatedTo` | `getAllocation` | `getScenario`

Topics

“Create and Manage Allocations”

Introduced in R2020b

systemcomposer.allocation.AllocationScenario

Class that represents allocation scenario

Description

The `systemcomposer.allocation.AllocationScenario` class 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 that scenario belongs to
allocation set object

Allocation set that scenario belongs to, 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

Examples

Allocate Architectures in a Tire Pressure Monitoring System

This example shows how to use allocations to analyze a tire pressure monitoring system.

Overview

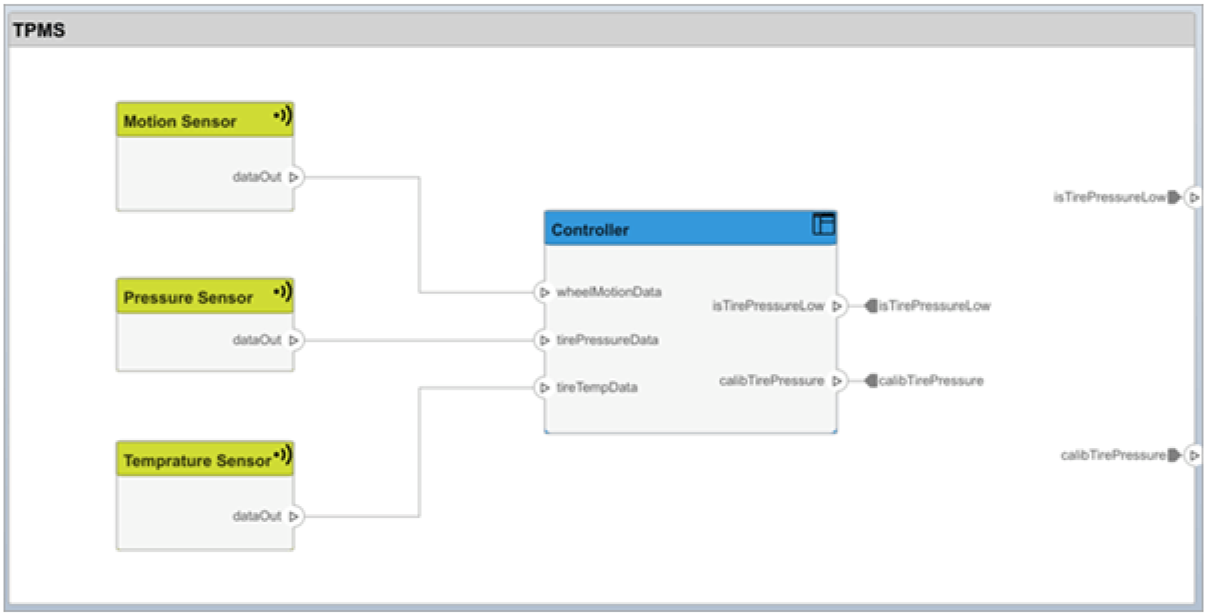
In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system 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.

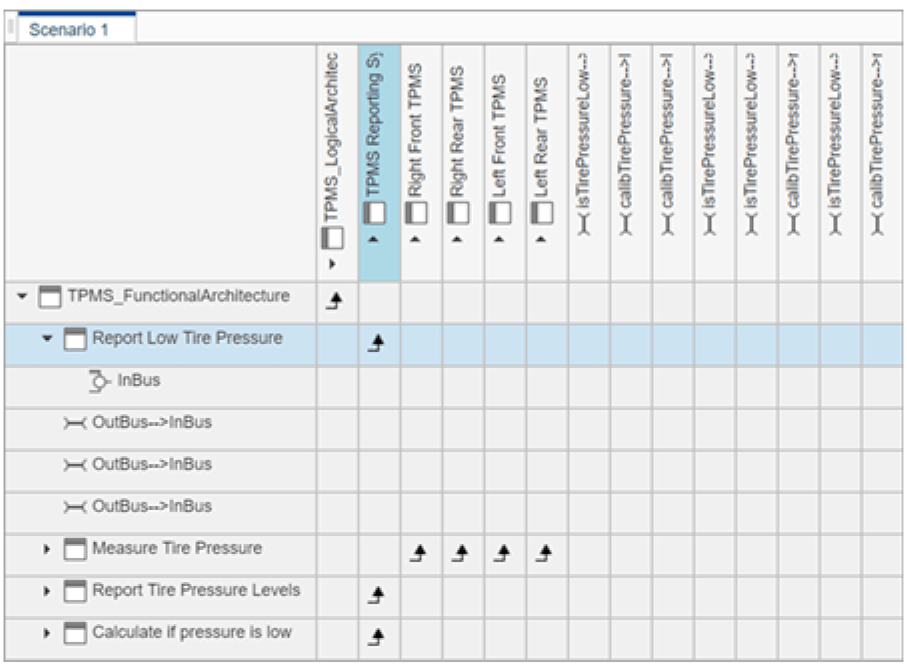
The allocation process is defined as linking these three architectures that fully describe the system. The linking captures the information about each architectural layer and makes it accessible to the others.

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 and the elements of `TPMS_LogicalArchitecture` are displayed in the first row. The arrows indicate the allocations between model elements.



This figure displays allocations in the architectural component level. 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 you can 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 example shows how to identify which functions will be provided by which suppliers using the specified allocations. The supplier information is stored in the logical model, since these are the components that the suppliers will be delivering to the system integrator.

```
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, nu
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.Supplie
        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
Report Low Tire Pressure	1	0	0	0
Measure temprature of tire	0	0	0	1
Calculate Tire Pressure	0	1	0	0
Measure rotations	0	1	0	0
Calculate if pressure is low	1	0	0	0
Report Tire Pressure Levels	1	0	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 of the components in the ECU, accumulate the binary size required for each of the allocated software components.

```
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.Bi
        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 a 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

Manage set of allocation scenarios

Description

The `AllocationSet` defines a collection of allocation scenarios between two 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 and/or target model, specified as a logical 1 (`true`) or 0 (`false`).

Data Types: `logical`

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

Whether the allocation set has unsaved changes, specified as a logical 1 (`true`) or 0 (`false`).

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
find	Find loaded allocation set
save	Save allocation set
close	Close allocation set
closeAll	Close all open allocation sets

Examples

Allocate Architectures in a Tire Pressure Monitoring System

This example shows how to use allocations to analyze a tire pressure monitoring system.

Overview

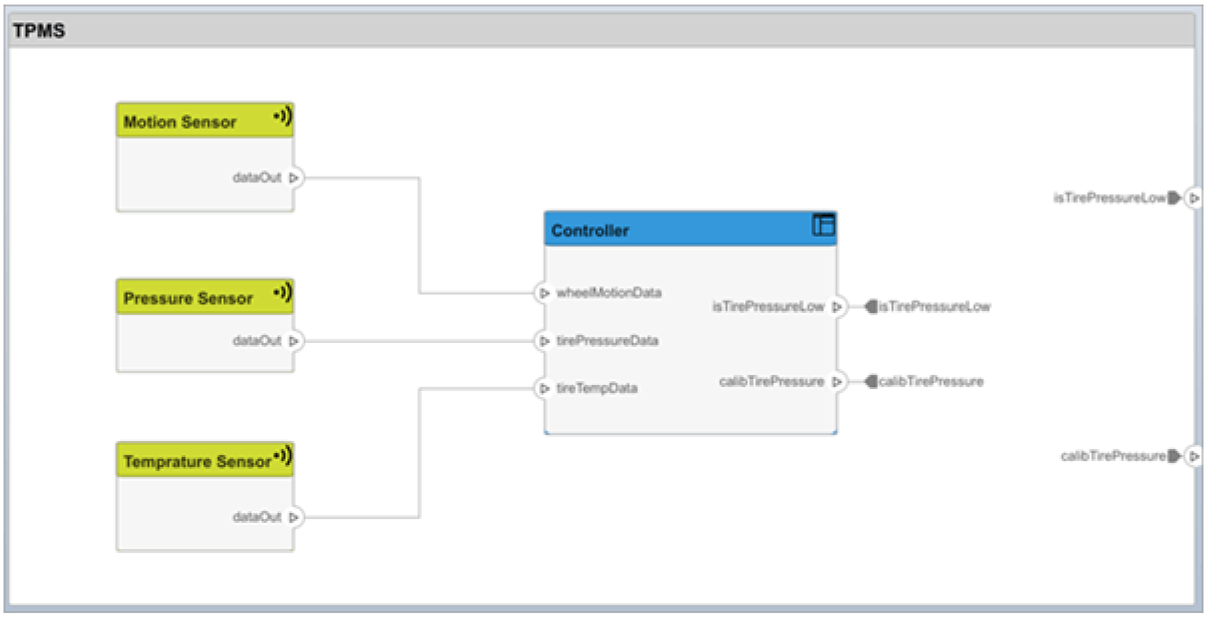
In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system 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.

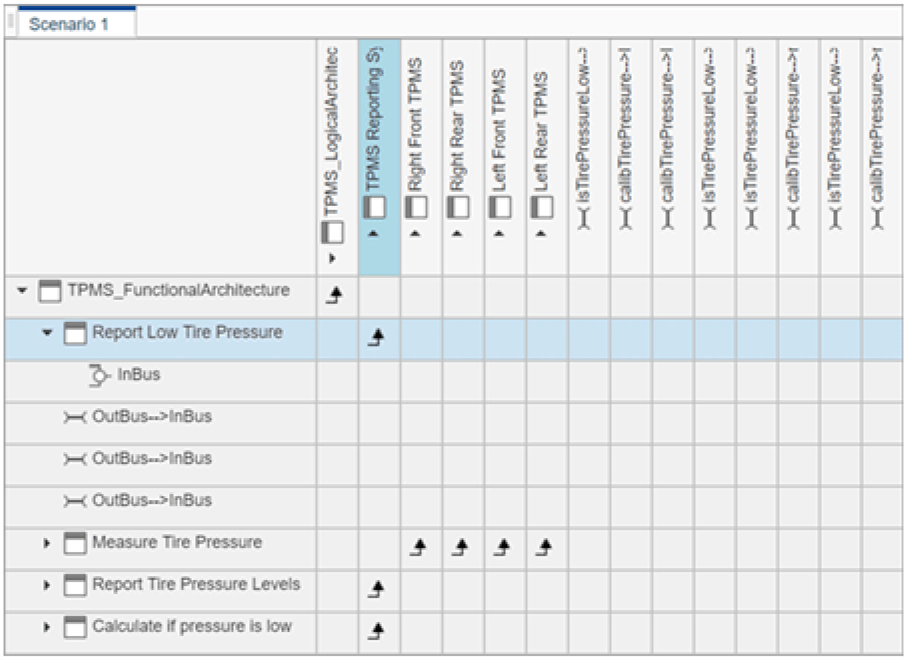
The allocation process is defined as linking these three architectures that fully describe the system. The linking captures the information about each architectural layer and makes it accessible to the others.

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 and the elements of `TPMS_LogicalArchitecture` are displayed in the first row. The arrows indicate the allocations between model elements.



This figure displays allocations in the architectural component level. 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 you can 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 example shows how to identify which functions will be provided by which suppliers using the specified allocations. The supplier information is stored in the logical model, since these are the components that the suppliers will be delivering to the system integrator.

```
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, nu
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.Supplie
        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
Report Low Tire Pressure	1	0	0	0
Measure temprature of tire	0	0	0	1
Calculate Tire Pressure	0	1	0	0
Measure rotations	0	1	0	0
Calculate if pressure is low	1	0	0	0
Report Tire Pressure Levels	1	0	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 of the components in the ECU, accumulate the binary size required for each of the allocated software components.

```
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.Bi
        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 a 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` | `editor` | `systemcomposer.allocation.Allocation` | `systemcomposer.allocation.AllocationScenario`

Topics

“Create and Manage Allocations”

Introduced in R2020b

systemcomposer.analysis.ArchitectureInstance

Class that represents architecture in analysis instance

Description

The `ArchitectureInstance` class 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 element in design model, specified as a `systemcomposer.arch.Architecture` object.

`NormalizeUnits` — Whether units are normalized
`true` or 1 | `false` or 0

Whether units normalize the value of properties in the instantiation, specified as a logical 1 (`true`) or 0 (`false`).

Data Types: `logical`

`IsStrict` — Whether instances only get properties if the instance's specification has the stereotype applied
`true` or 1 | `false` or 0

Whether instances only get properties if the instance's specification has the stereotype applied, specified as a logical 1 (`true`) or 0 (`false`).

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
`systemcomposer.IteratorDirection.TopDown` |
`systemcomposer.IteratorDirection.BottomUp` |
`systemcomposer.IteratorDirection.PreOrder` |
`systemcomposer.IteratorDirection.PostOrder`

Analysis direction, specified as an enumeration.

Example: `'TopDown'`

Example: `'PreOrder'`

Example: `'PostOrder'`

Example: `'BottomUp'`

Data Types: `enum`

`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 is updated automatically when the design model changes
`true` or 1 | `false` or 0

Whether analysis viewer is updated automatically when the design model changes, specified as a logical 1 (`true`) or 0 (`false`).

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

Analysis of Latency Characteristics

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

Create a 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('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','DefaultValue','true');
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','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('LatencyProfile'); % 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','LatencyProfile.NodeLatency');
batchApplyStereotype(arch,'Port','LatencyProfile.PortLatency');
batchApplyStereotype(arch,'Connector','LatencyProfile.ConnectorLatency');

% Instantiate using the analysis function
instance = instantiate(model.Architecture,'LatencyProfile','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('LatencyProfile.NodeLatency.resources')
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue('LatencyProfile.ConnectorLatency.secure')
```

```
defaultSecure = logical
   1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue('LatencyProfile.PortLatency.queueDep
```

```
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

This example shows how to model a typical automotive electrical system as an architectural model and run primitive analysis. The elements in the model can be broadly grouped as either source or load. Various properties of the sources and loads are set as part of the stereotype. The 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 the Model

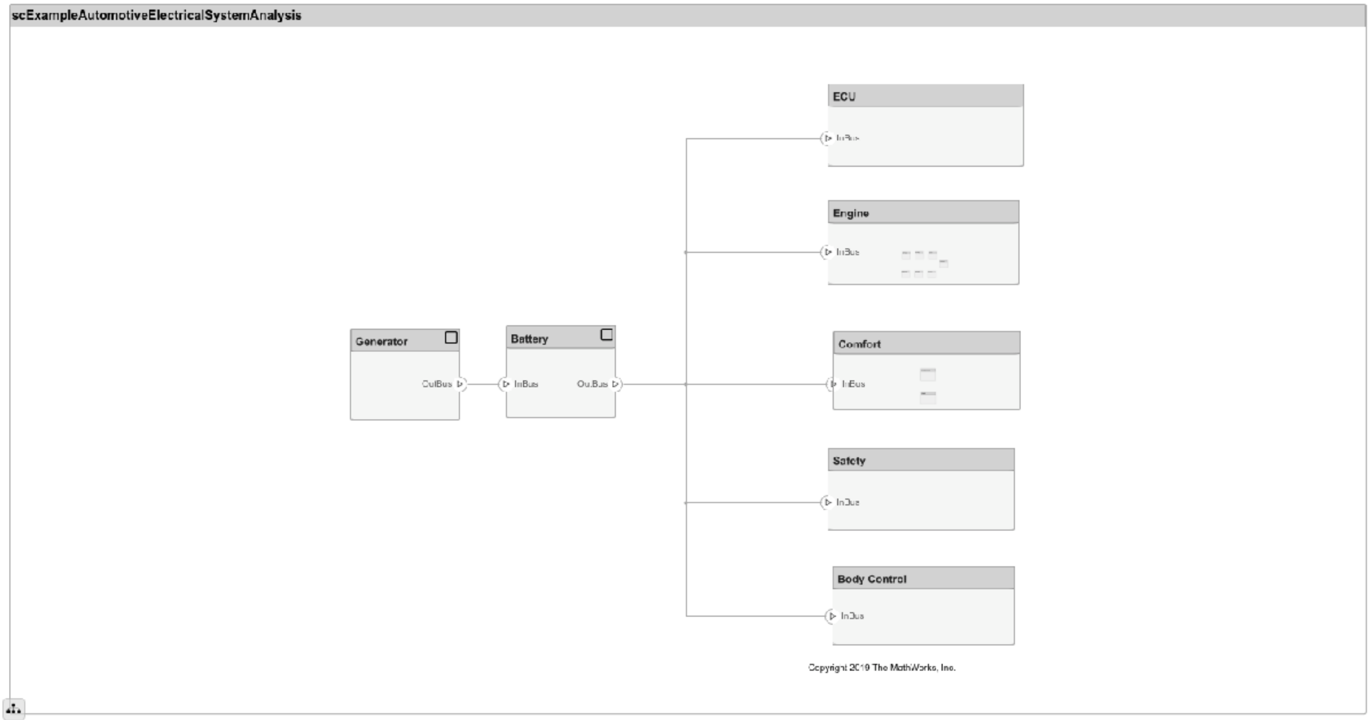
The generator charges the battery while the engine is running. The battery, along with the generator supports the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties 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 the Model and Run the 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 the 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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	"Analyze Architecture"

Term	Definition	Application	More Information
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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.ConnectorInstance` | `systemcomposer.analysis.Instance` | `systemcomposer.analysis.PortInstance`

Topics

“Write Analysis Function”

Introduced in R2019a

systemcomposer.analysis.ComponentInstance

Class that represents component in analysis instance

Description

The `ComponentInstance` class 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 within the architecture.

`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, connecting child components, specified as an array of `systemcomposer.analysis.ConnectorInstance` objects.

`Parent` — Parent of the component
architecture instance object

Parent of the 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

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

Analysis of Latency Characteristics

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

Create a 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('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','DefaultValue','true');
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','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('LatencyProfile'); % 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','LatencyProfile.NodeLatency');
batchApplyStereotype(arch,'Port','LatencyProfile.PortLatency');
batchApplyStereotype(arch,'Connector','LatencyProfile.ConnectorLatency');

% Instantiate using the analysis function
instance = instantiate(model.Architecture,'LatencyProfile','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('LatencyProfile.NodeLatency.resources')
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue('LatencyProfile.ConnectorLatency.secure')
```

```
defaultSecure = logical
   1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue('LatencyProfile.PortLatency.queueDep
```

```
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

This example shows how to model a typical automotive electrical system as an architectural model and run primitive analysis. The elements in the model can be broadly grouped as either source or load. Various properties of the sources and loads are set as part of the stereotype. The 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 the Model

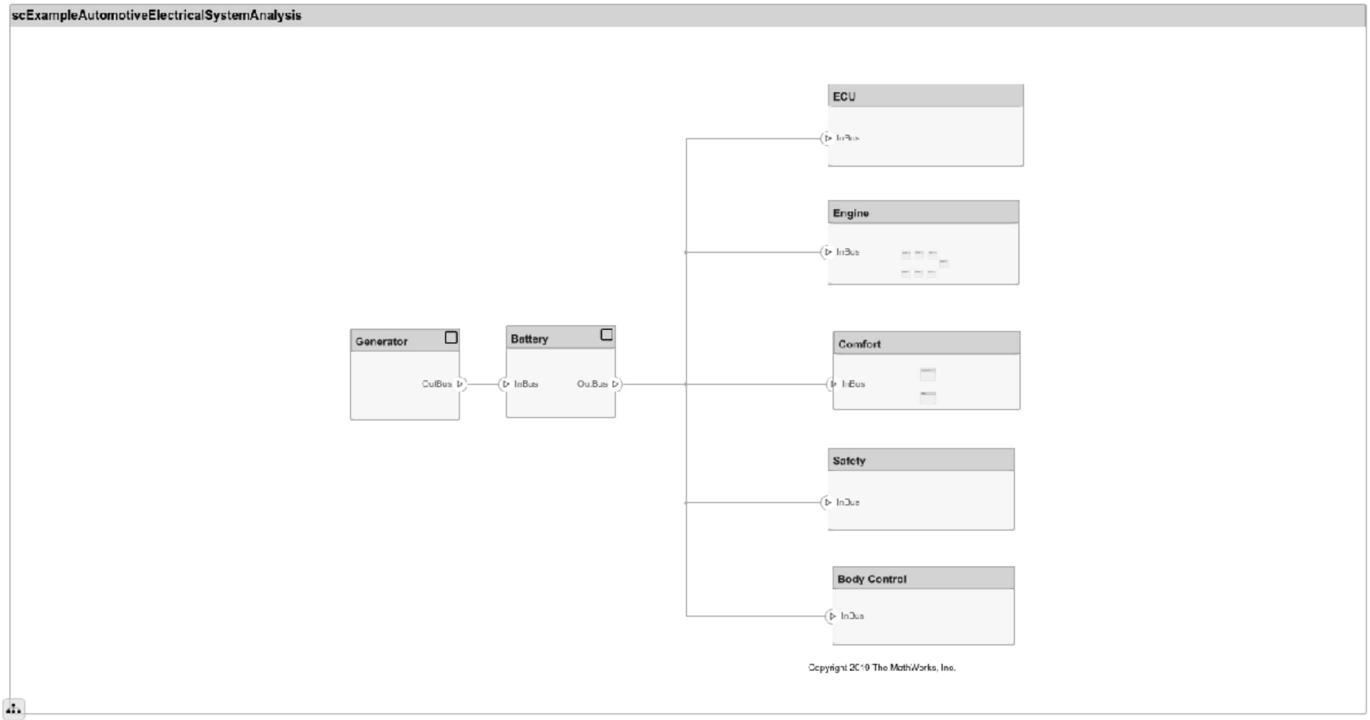
The generator charges the battery while the engine is running. The battery, along with the generator supports the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties 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 the Model and Run the 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 the 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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	“Analyze Architecture”

Term	Definition	Application	More Information
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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` | `iterate` | `loadInstance` | `refresh` | `save` | `systemcomposer.analysis.ArchitectureInstance` | `systemcomposer.analysis.ConnectorInstance` | `systemcomposer.analysis.Instance` | `systemcomposer.analysis.PortInstance` | `update`

Topics

“Write Analysis Function”

Introduced in R2019a

systemcomposer.analysis.ConnectorInstance

Class that represents connector in analysis instance

Description

The `ConnectorInstance` class 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

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

Analysis of Latency Characteristics

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

Create a 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('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','DefaultValue','true');
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','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('LatencyProfile'); % 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','LatencyProfile.NodeLatency');
batchApplyStereotype(arch,'Port','LatencyProfile.PortLatency');
batchApplyStereotype(arch,'Connector','LatencyProfile.ConnectorLatency');

% Instantiate using the analysis function
instance = instantiate(model.Architecture,'LatencyProfile','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('LatencyProfile.NodeLatency.resources')
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue('LatencyProfile.ConnectorLatency.secure')
```

```
defaultSecure = logical
   1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue('LatencyProfile.PortLatency.queueDep
```

```
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

This example shows how to model a typical automotive electrical system as an architectural model and run primitive analysis. The elements in the model can be broadly grouped as either source or load. Various properties of the sources and loads are set as part of the stereotype. The 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 the Model

The generator charges the battery while the engine is running. The battery, along with the generator supports 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:

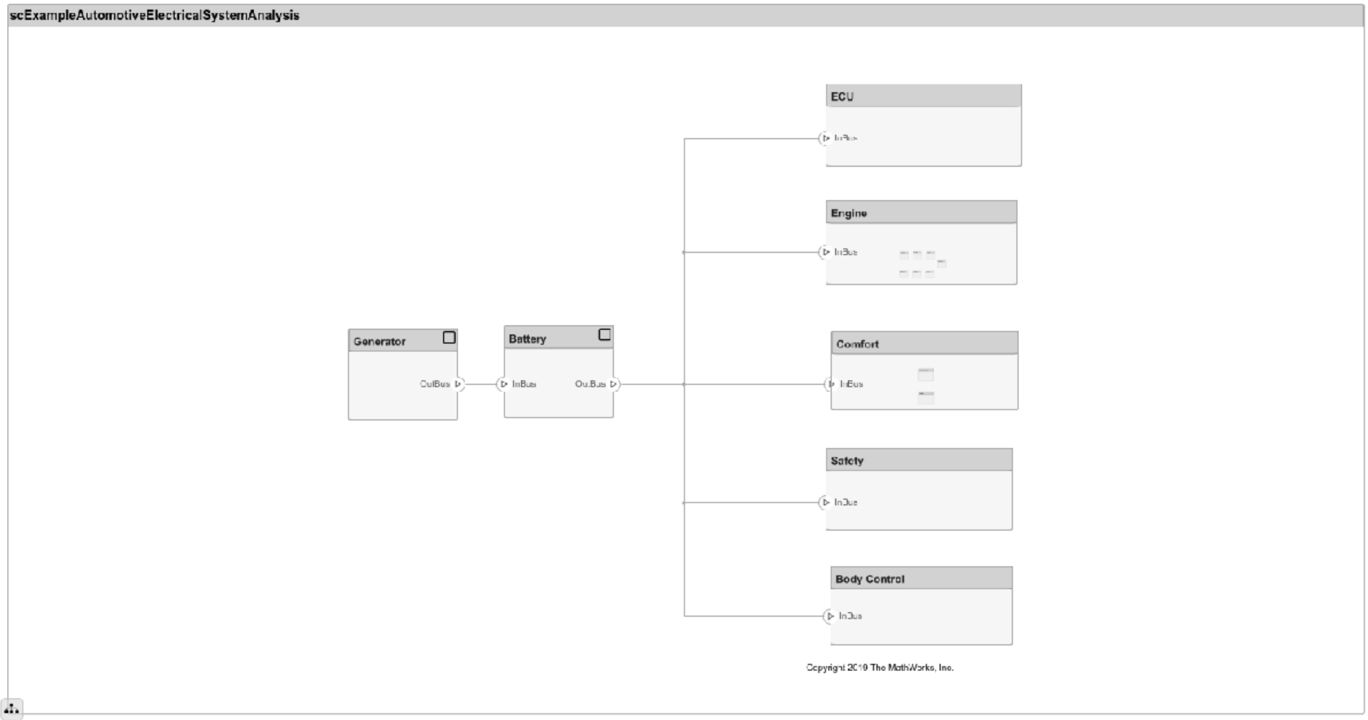
- Total `KeyOffLoad`.
- Number of days required for `KeyOffLoad` to discharge 30% of the battery.
- Total `CrankingInRush` current.
- Total `Cranking` current.
- Ability of the battery to start the vehicle at 0°F based on the battery cold cranking amps (CCA). The discharge time is computed based on Puekert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

Load the Model and Run the 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 the 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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	“Analyze Architecture”

Term	Definition	Application	More Information
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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` | `iterate` | `loadInstance` | `refresh` | `save` | `systemcomposer.analysis.ArchitectureInstance` | `systemcomposer.analysis.ComponentInstance` | `systemcomposer.analysis.Instance` | `systemcomposer.analysis.PortInstance` | `update`

Topics

"Write Analysis Function"

Introduced in R2019a

systemcomposer.analysis.Instance

Class that represents model element in analysis instance

Description

The `Instance` class represents an instance of a model element.

Related classes 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

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

Analysis of Latency Characteristics

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

Create a 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('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','DefaultValue','true');
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','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('LatencyProfile'); % 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','LatencyProfile.NodeLatency');
```

```
batchApplyStereotype(arch,'Port','LatencyProfile.PortLatency');
batchApplyStereotype(arch,'Connector','LatencyProfile.ConnectorLatency');

% Instantiate using the analysis function
instance = instantiate(model.Architecture,'LatencyProfile','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('LatencyProfile.NodeLatency.resources')
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue('LatencyProfile.ConnectorLatency.secure')
```

```
defaultSecure = logical
   1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue('LatencyProfile.PortLatency.queueDep
```

```
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

This example shows how to model a typical automotive electrical system as an architectural model and run primitive analysis. The elements in the model can be broadly grouped as either source or load. Various properties of the sources and loads are set as part of the stereotype. The 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 the Model

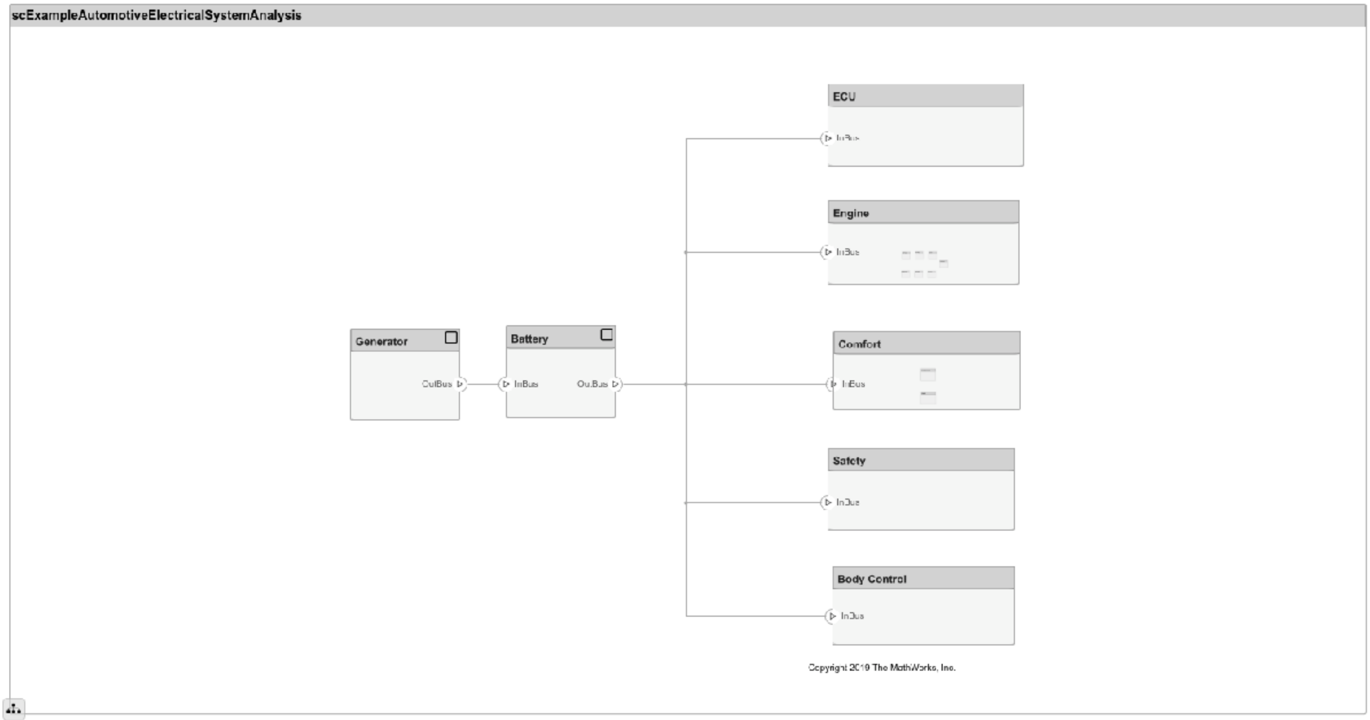
The generator charges the battery while the engine is running. The battery, along with the generator supports the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties 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 the Model and Run the 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 the 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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	“Analyze Architecture”

Term	Definition	Application	More Information
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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` | `iterate` | `loadInstance` | `refresh` | `save` | `systemcomposer.analysis.ArchitectureInstance` | `systemcomposer.analysis.ComponentInstance` | `systemcomposer.analysis.ConnectorInstance` | `systemcomposer.analysis.PortInstance` | `update`

Topics

"Write Analysis Function"

Introduced in R2019a

systemcomposer.analysis.PortInstance

Class that represents port in analysis instance

Description

The `PortInstance` class 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 the port
component instance object

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

Specification — Reference to port in the design model
base port object

Reference to port in the 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

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

Analysis of Latency Characteristics

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

Create a 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('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','DefaultValue','true');
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','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('LatencyProfile'); % 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','LatencyProfile.NodeLatency');
batchApplyStereotype(arch,'Port','LatencyProfile.PortLatency');
batchApplyStereotype(arch,'Connector','LatencyProfile.ConnectorLatency');

% Instantiate using the analysis function
instance = instantiate(model.Architecture,'LatencyProfile','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('LatencyProfile.NodeLatency.resources')
```

```
defaultResources = 1
```

```
defaultSecure = instance.Connectors(1).getValue('LatencyProfile.ConnectorLatency.secure')
```

```
defaultSecure = logical
   1
```

```
defaultQueueDepth = instance.Components(1).Ports(1).getValue('LatencyProfile.PortLatency.queueDe
```

```
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

This example shows how to model a typical automotive electrical system as an architectural model and run primitive analysis. The elements in the model can be broadly grouped as either source or load. Various properties of the sources and loads are set as part of the stereotype. The 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 the Model

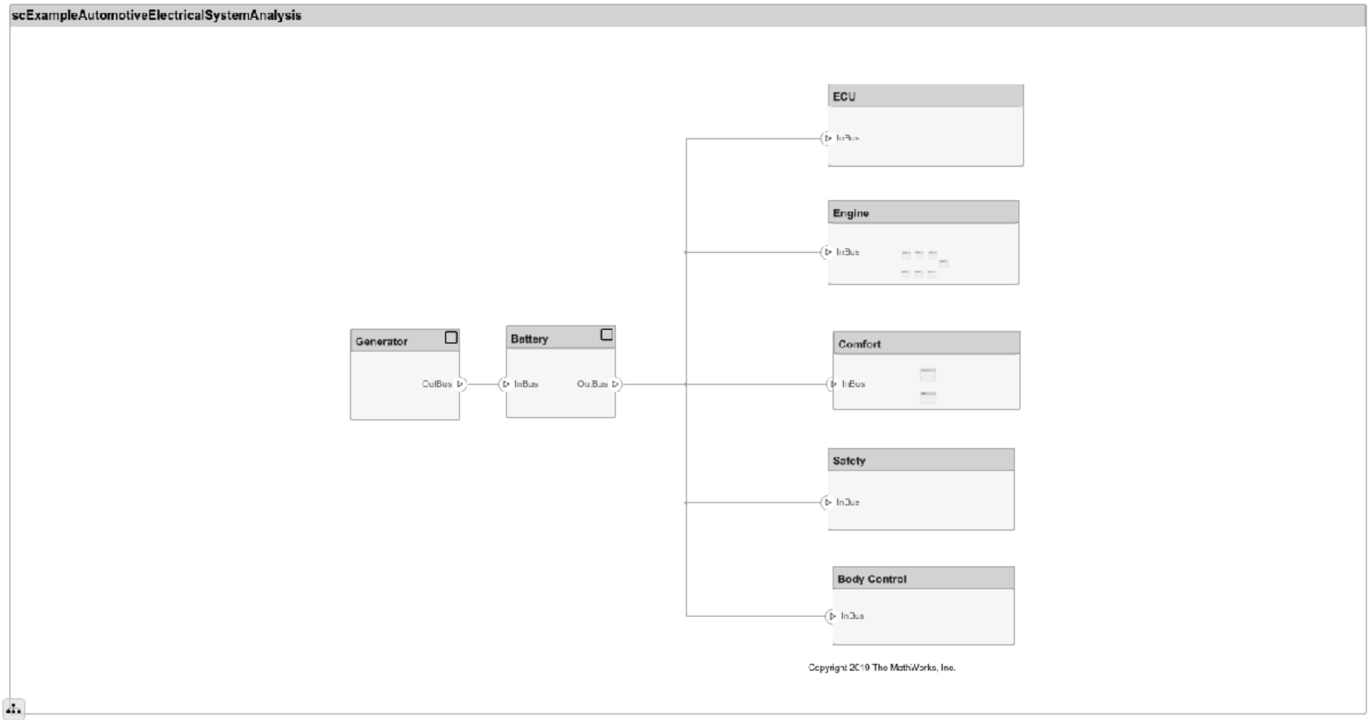
The generator charges the battery while the engine is running. The battery, along with the generator supports the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties 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 the Model and Run the 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 the 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 analysis to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.	“Analyze Architecture”

Term	Definition	Application	More Information
instance	An instance is an occurrence of an architecture model at a given point of time.	You can update an instance with changes to a model, but the instance will not update with changes in active variants or model references. You can use an instance, saved in an `.MAT` file, of a System Composer architecture model for analysis.	"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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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` | `iterate` | `loadInstance` | `refresh` | `save` | `systemcomposer.analysis.ArchitectureInstance` | `systemcomposer.analysis.ComponentInstance` | `systemcomposer.analysis.ConnectorInstance` | `systemcomposer.analysis.Instance` | `update`

Topics

“Write Analysis Function”

Introduced in R2019a

systemcomposer.arch.Architecture

Class that represents architecture in model

Description

The `Architecture` class represents an architecture in the 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
`composition` | `behavior` | `view`

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

Data Types: `ArchitectureDefinition 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 of architecture, specified as an array of `systemcomposer.arch.ArchitecturePort` objects.

`Connectors` — Connectors that connect child components of this architecture
array of connector objects

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

`ExternalUID` — Unique external identifier
character vector

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

Data Types: `char`

`Model` — Parent System Composer model
model object

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

`SimulinkHandle` — Simulink handle
numeric value

Simulink handle for architecture, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

`SimulinkModelHandle` — Simulink handle to parent System Composer model
numeric value

Simulink handle to parent model of architecture, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

Object Functions

addComponent	Add components to architecture
addVariantComponent	Add variant components to architecture
addPort	Add ports to architecture
connect	Create architecture model connections
applyStereotype	Apply stereotype to architecture model element

getStereotypes	Get stereotypes applied on element of architecture model
removeStereotype	Remove stereotype from model element
batchApplyStereotype	Apply stereotype to all elements in architecture
iterate	Iterate over model elements
instantiate	Create analysis instance from specification
setProperty	Set property value corresponding to stereotype applied to element
getProperty	Get property value corresponding to stereotype applied to element
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from component
getStereotypeProperties	Get stereotype property names on element
removeProfile	Remove profile from model
applyProfile	Apply profile to model

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'i
planningPorts(2).setInterface(interface);
```

```
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

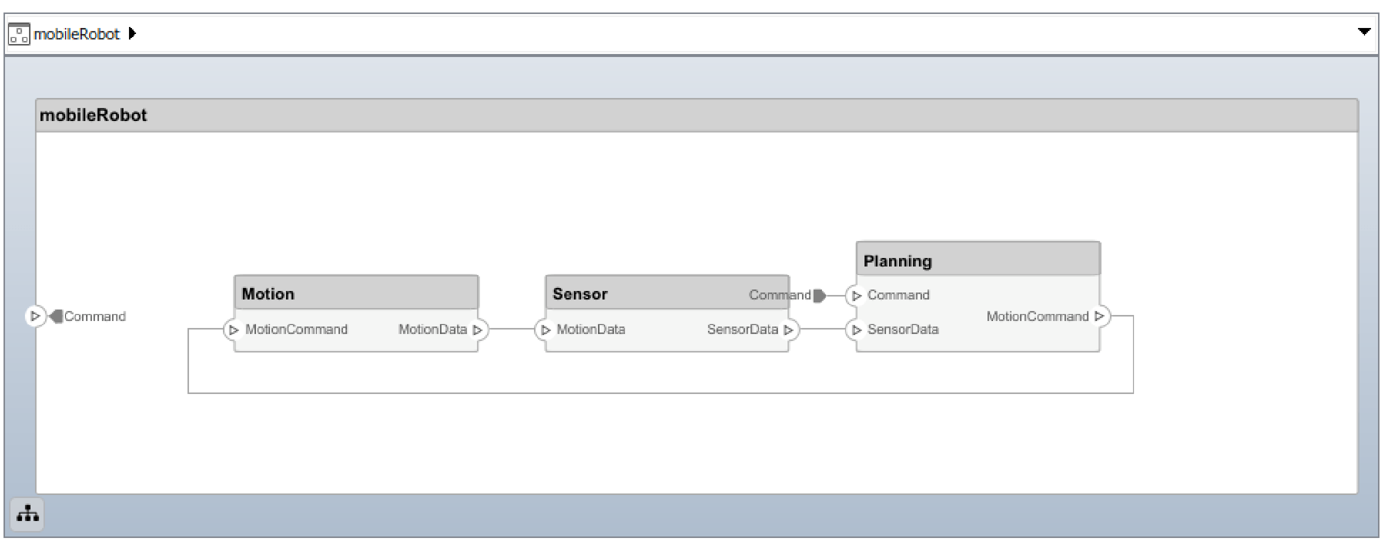
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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 = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

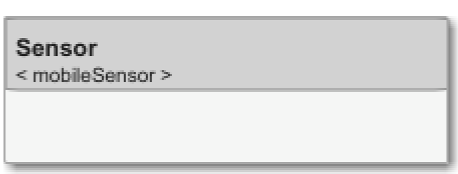
```
Simulink.BlockDiagram.arrangeSystem('mobileRobotAPI/Motion');
```

Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

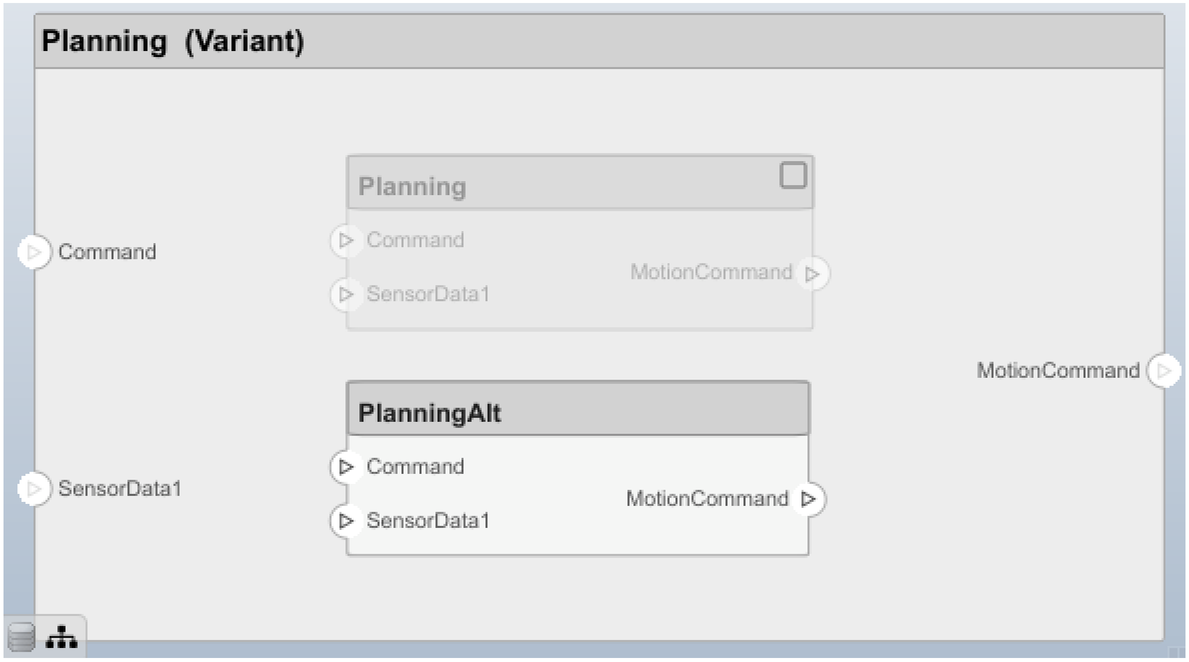
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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"

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>port</td><td>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.</td><td>There are different types of ports:
Component ports are interaction points on the component to other components.
Architecture ports are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</td><td>“Ports”</td></tr>
<tr><td>connector</td><td>Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.</td><td>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.</td><td>“Connections”</td></tr>
</table>

See Also

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

Topics

“Create an Architecture Model”

Introduced in R2019a

systemcomposer.arch.ArchitecturePort

Class that represents input and output ports of architecture

Description

The `ArchitecturePort` class represents the input and output ports of an 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'`

Port direction, specified as a character vector with values `'Input'` and `'Output'`.

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
signal interface object

Interface associated with port, specified as a `systemcomposer.interface.SignalInterface` object.

`Connectors` — Port connectors
array of connector objects

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

`Connected` — Whether port has connections
`true` or 1 | `false` or 0

Whether port has connections, specified as a logical 1 (`true`) or 0 (`false`).

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`

`ExternalUID` — Unique external identifier
character vector

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

Data Types: `char`

`Model` — Parent System Composer model
model object

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

`SimulinkHandle` — Simulink handle
numeric value

Simulink handle for architecture port, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

`SimulinkModelHandle` — Simulink handle to parent System Composer model
numeric value

Simulink handle to parent model of architecture port, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirement 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
createAnonymousInterface	Create and set anonymous 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
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);
```

```
planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);
```

```
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

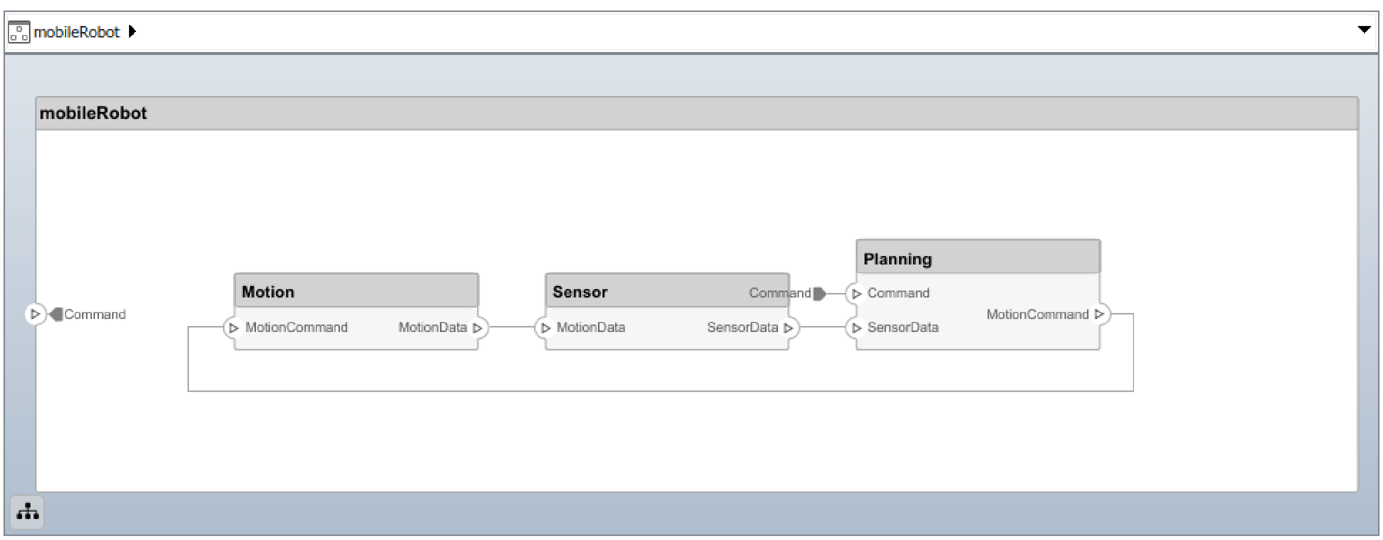
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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 = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

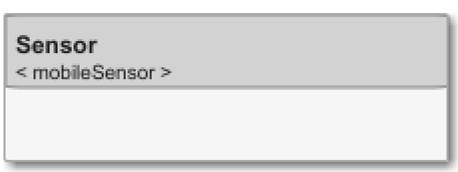
```
Simulink.BlockDiagram.arrangeSystem('mobileRobotAPI/Motion');
```

Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

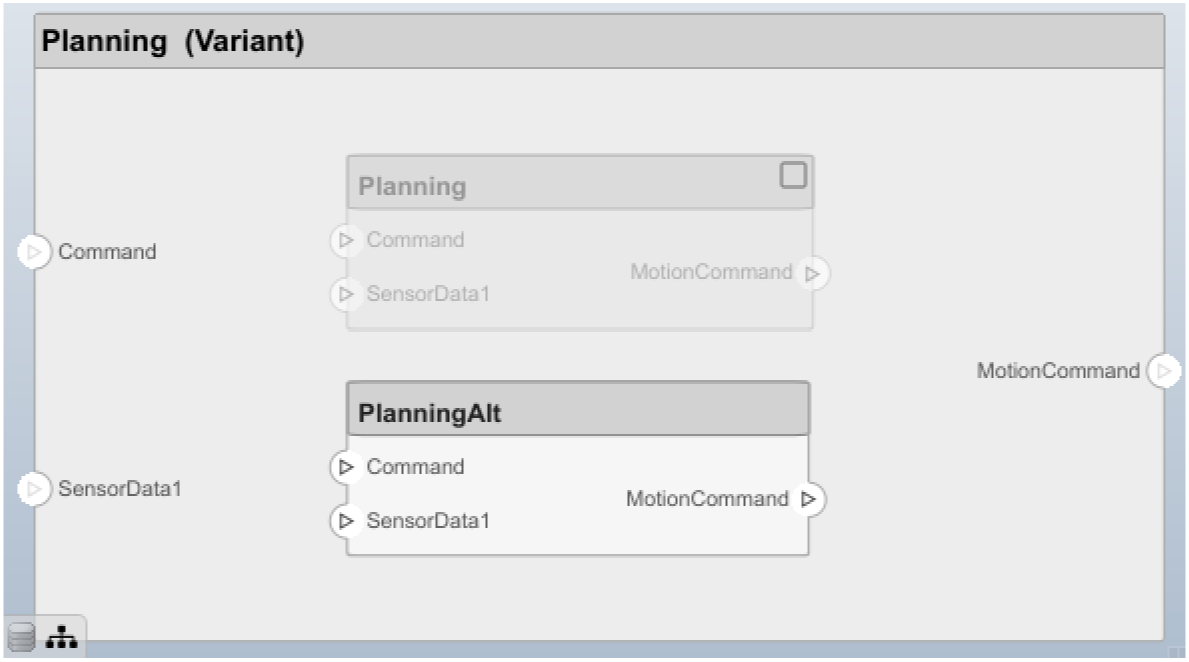
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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 | `addPort` | `systemcomposer.arch.BasePort` | `systemcomposer.arch.ComponentPort` | `systemcomposer.arch.Element`

Topics

"Create an Architecture Model"

Introduced in R2019a

systemcomposer.arch.BaseComponent

Common base class for all components in architecture model

Description

A `systemcomposer.arch.BaseComponent` cannot be constructed. Either create a `systemcomposer.arch.Component` or `systemcomposer.arch.VariantComponent`. This 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 element and through all operations that preserve the `UUID`.

Data Types: `char`

`Model` — Parent System Composer model
model object

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

`SimulinkHandle` — Simulink handle
numeric value

Simulink handle for component, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

`SimulinkModelHandle` — Simulink handle to parent System Composer model
numeric value

Simulink handle to parent model of component, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

Object Functions

getProperty	Get property value corresponding to stereotype applied to element
setProperty	Set property value corresponding to stereotype applied to element
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from component
getStereotypeProperties	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
isReference	Find if component is reference to another model
connect	Create architecture model connections
getPort	Get port from component
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'i
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

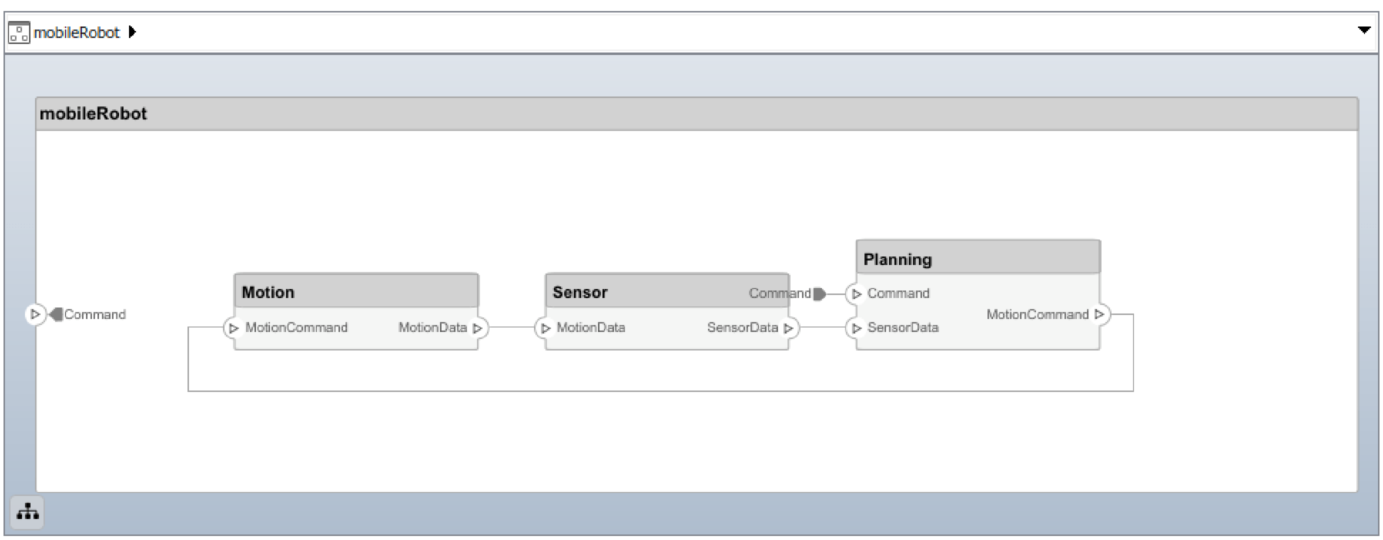
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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 = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

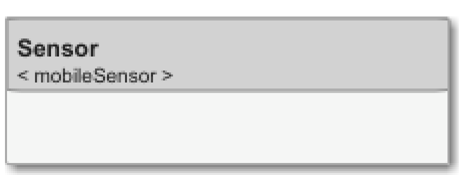
```
Simulink.BlockDiagram.arrangeSystem('mobileRobotAPI/Motion');
```

Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
```

```
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

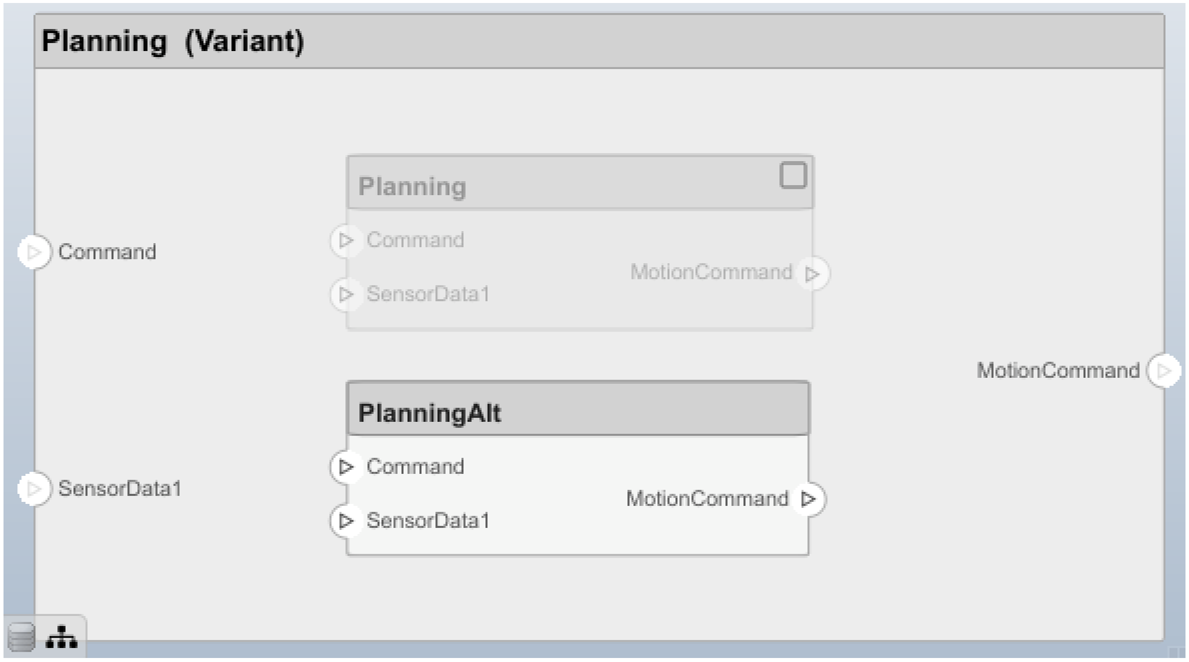
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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.Component` | `systemcomposer.arch.Element` | `systemcomposer.arch.VariantComponent`

Topics

"Create an Architecture Model"

Introduced in R2019b

systemcomposer.arch.BasePort

Common base class for all ports in architecture model

Description

A `systemcomposer.arch.BasePort` cannot be constructed. Either create a `systemcomposer.arch.ArchitecturePort` or a `systemcomposer.arch.ComponentPort`. This 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'`

Port direction, specified as a character vector with values `'Input'` and `'Output'`.

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
signal interface object

Interface associated with port, specified as a `systemcomposer.interface.SignalInterface` object.

`Connectors` — Port connectors
array of connector objects

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

`Connected` — Whether port has connections
`true` or 1 | `false` or 0

Whether port has connections, specified as a logical 1 (`true`) or 0 (`false`).

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`

`ExternalUID` — Unique external identifier
character vector

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

Data Types: `char`

`Model` — Parent System Composer model
model object

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

`SimulinkHandle` — Simulink handle
numeric value

Simulink handle for port, specified as a `double`. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

`SimulinkModelHandle` — Simulink handle to parent System Composer model
numeric value

Simulink handle to parent model of port, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

Object Functions

getProperty	Get property value corresponding to stereotype applied to element
setProperty	Set property value corresponding to stereotype applied to element
getPropertyValue	Get value of architecture property
getEvaluatedPropertyValue	Get evaluated value of property from component
getStereotypeProperties	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
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'i
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

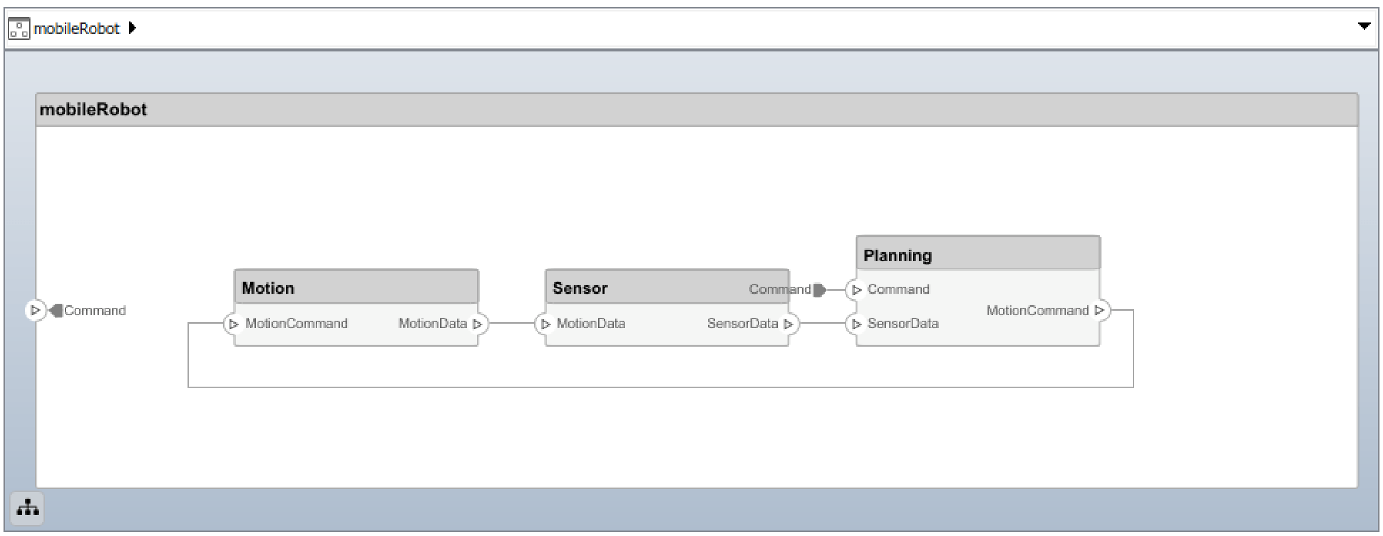
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
```

```
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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 = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

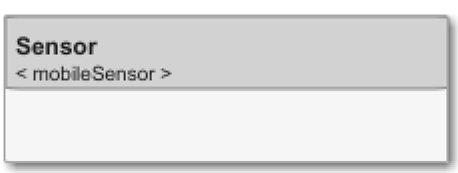
```
Simulink.BlockDiagram.arrangeSystem('mobileRobotAPI/Motion');
```

Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
```

```
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

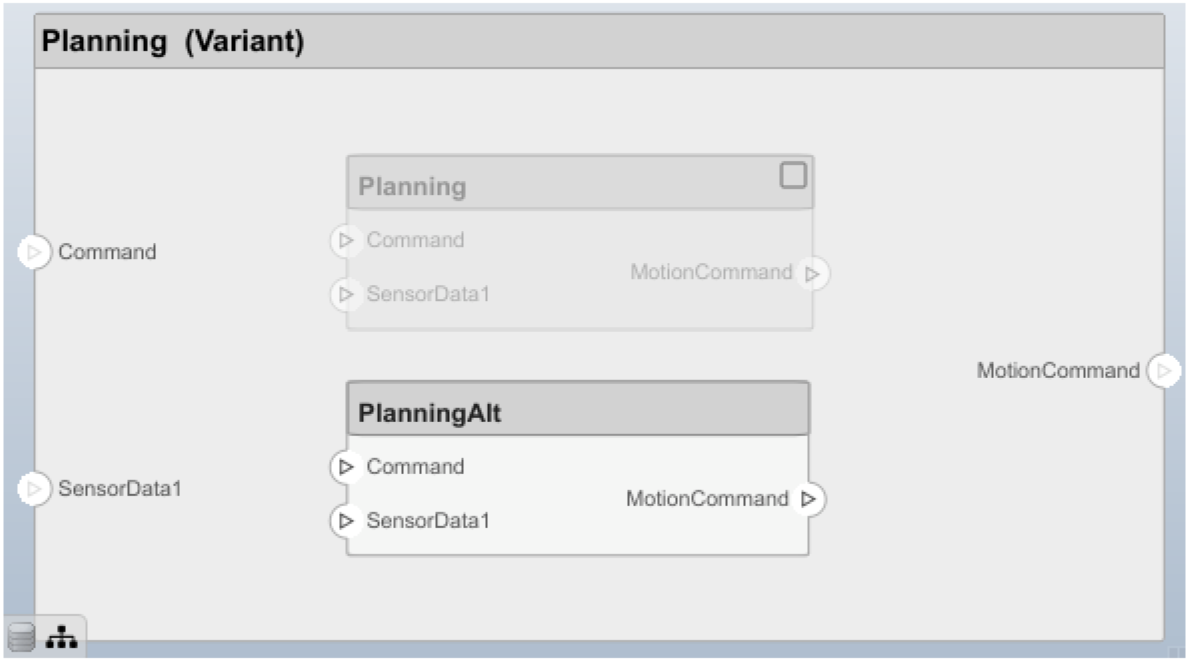
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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` | `systemcomposer.arch.Element`

Topics

"Create an Architecture Model"

Introduced in R2019a

systemcomposer.arch.Component

Class that represents component

Description

The `Component` class represents a component in an architecture 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` — Handle to parent architecture that owns component
architecture object

Handle to parent 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`

`ExternalUID` — Unique external identifier
character vector

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

Data Types: `char`

`Model` — Parent System Composer model
model object

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

`SimulinkHandle` — Simulink handle
numeric value

Simulink handle for component, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

`SimulinkModelHandle` — Simulink handle to parent System Composer model
numeric value

Simulink handle to parent model of component, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

Object Functions

saveAsModel	Save architecture of component to separate model
createSimulinkBehavior	Create Simulink behavior and link to component
createStateflowChartBehavior	Add Stateflow chart behavior to component
linkToModel	Link component to a model
inlineComponent	Inline reference architecture or behavior into model
makeVariant	Convert component to variant choice
isReference	Find if component is reference to another model
connect	Create architecture model connections
getPort	Get port from component
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
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

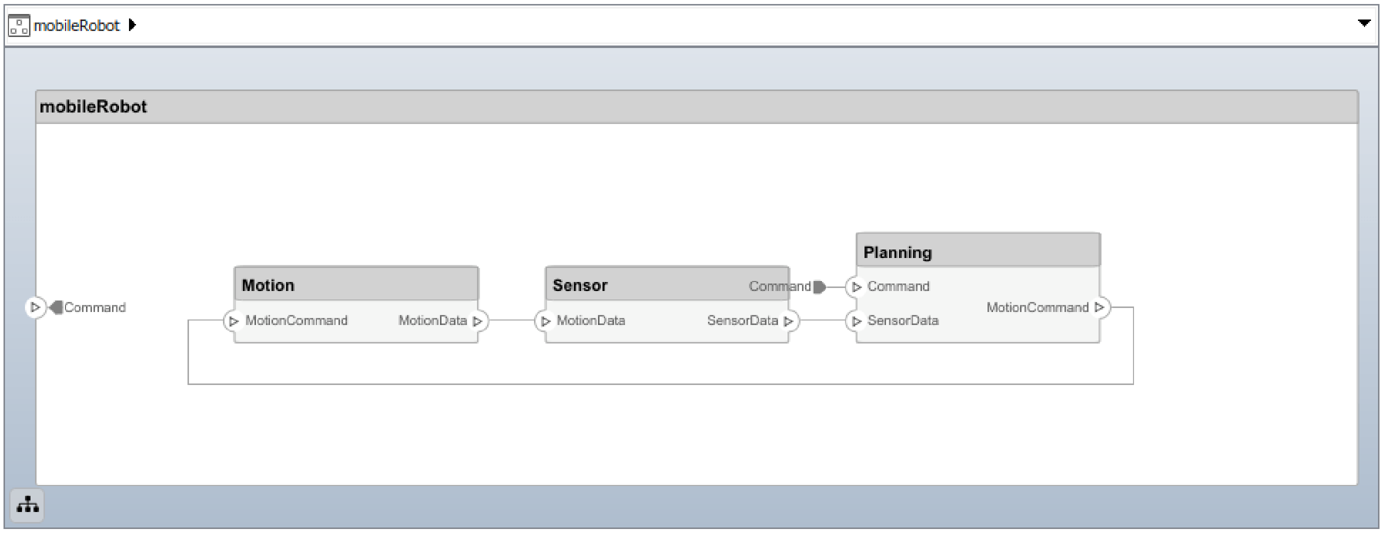
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all sen
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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 = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

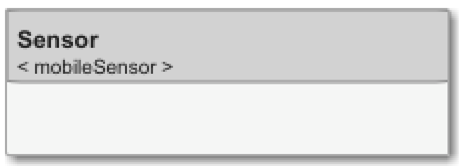
```
Simulink.BlockDiagram.arrangeSystem('mobileRobotAPI/Motion');
```

Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

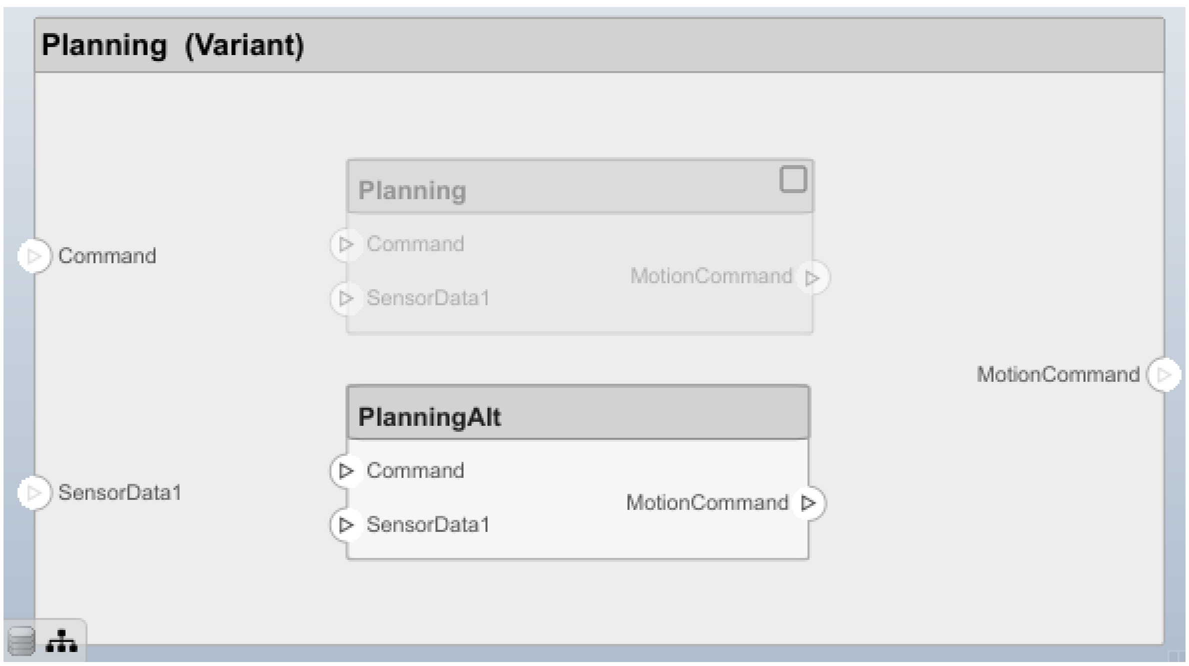
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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”

<table>
<tr><th>Term</th><th>Definition</th><th>Application</th><th>More Information</th></tr>
<tr><td>port</td><td>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.</td><td>There are different types of ports:
• Component ports are interaction points on the component to other components.
• Architecture ports are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.</td><td>“Ports”</td></tr>
<tr><td>connector</td><td>Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures.</td><td>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.</td><td>“Connections”</td></tr>
</table>

See Also

Component | `addComponent` | `createModel` | `systemcomposer.arch.Architecture` | `systemcomposer.arch.Element`

Topics

“Create an Architecture Model”

Introduced in R2019a

systemcomposer.arch.ComponentPort

Class that represents input and output ports of component

Description

The `ComponentPort` class represents the input and output ports of a 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.

```
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'`

Port direction, specified as a character vector with values `'Input'` and `'Output'`.

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
signal interface object

Interface associated with port, specified as a `systemcomposer.interface.SignalInterface` object.

`Connectors` — Port connectors
array of connector objects

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

`Connected` — Whether port has connections
`true` or 1 | `false` or 0

Whether port has connections, specified as a logical 1 (`true`) or 0 (`false`).

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`

`ExternalUID` — Unique external identifier
character vector

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

Data Types: `char`

`Model` — Parent System Composer model
model object

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

`SimulinkHandle` — Simulink handle
numeric value

Simulink handle for component port, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

`SimulinkModelHandle` — Simulink handle to parent System Composer model
numeric value

Simulink handle to parent model of component port, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

Object Functions

setName	Set name for port
setInterface	Set interface for port
createAnonymousInterface	Create and set anonymous 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
connect	Create architecture model connections
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
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
```

```
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

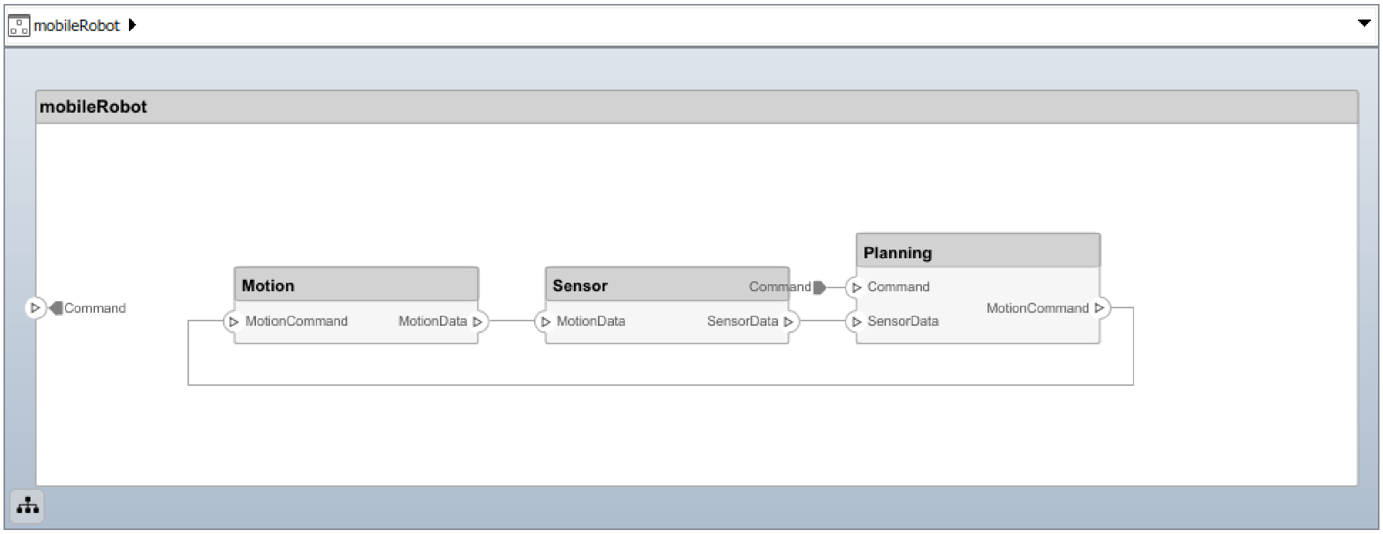
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

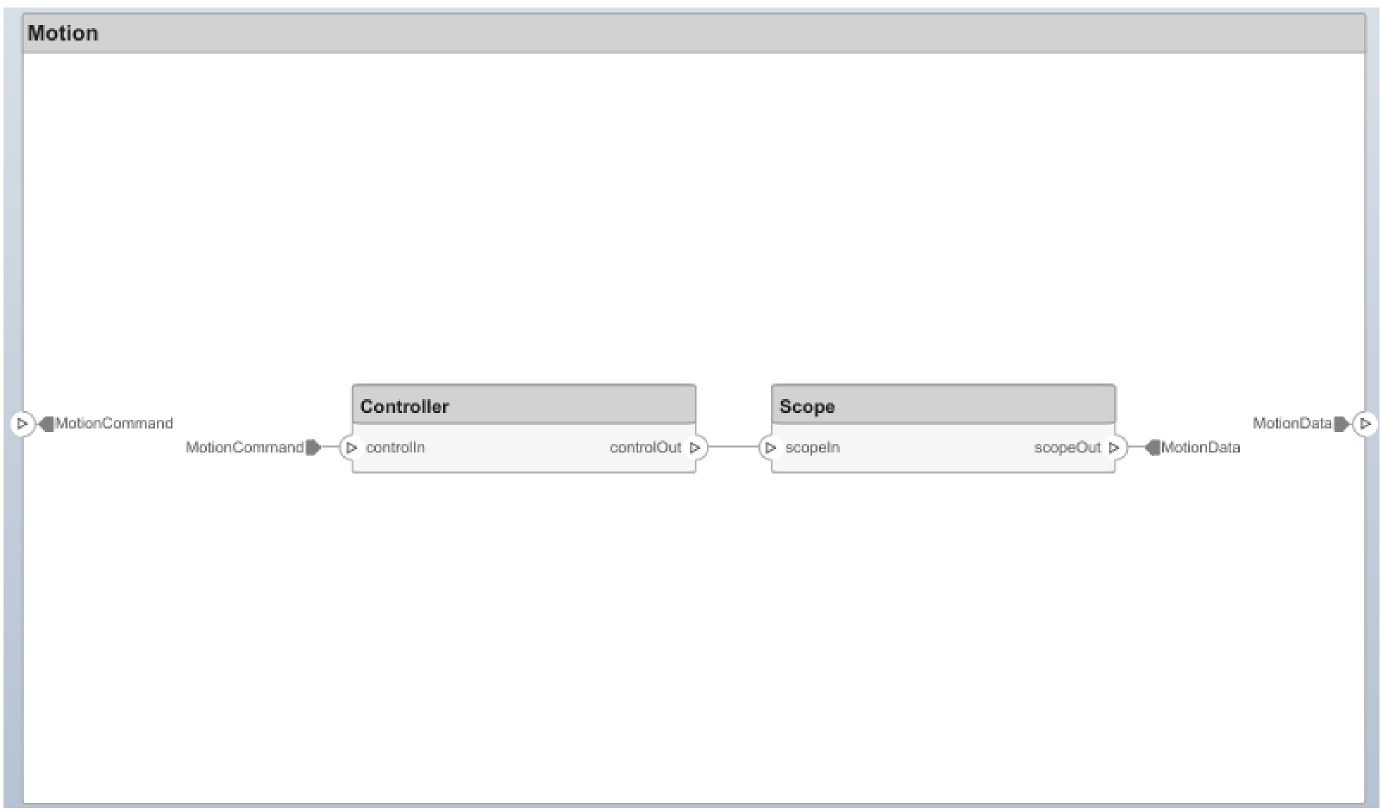
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



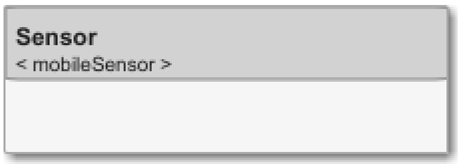
Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the Planning component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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.

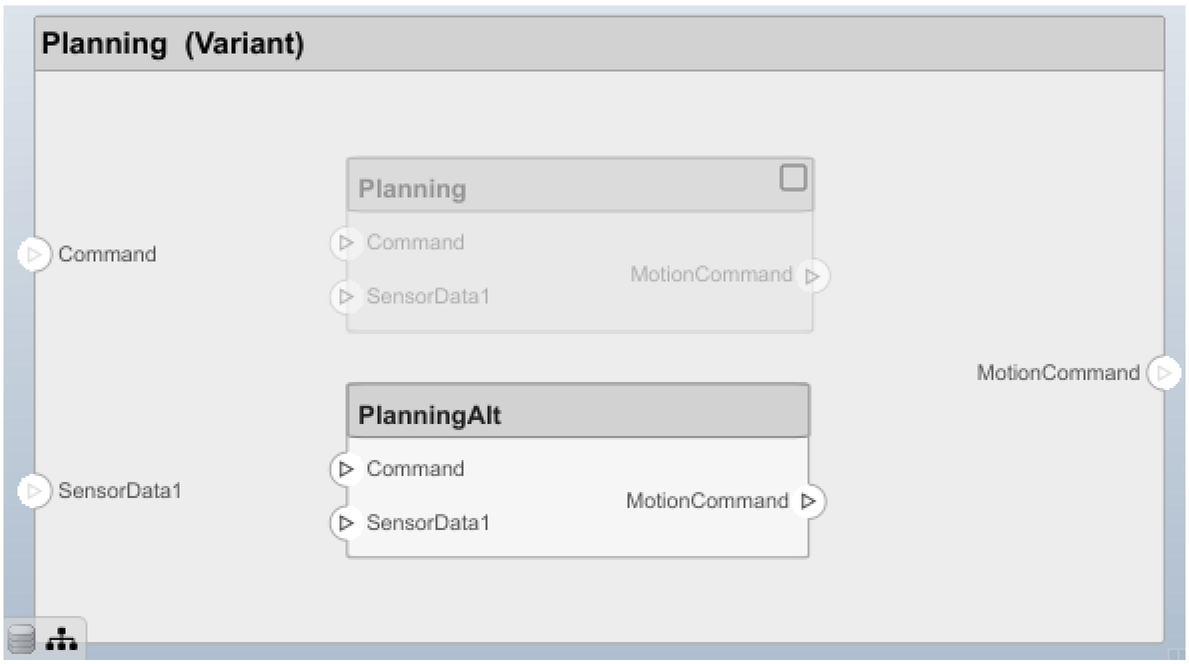
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 | `addPort` | `getPort` | `systemcomposer.arch.ArchitecturePort` | `systemcomposer.arch.BasePort` | `systemcomposer.arch.Element`

Topics

“Create an Architecture Model”

Introduced in R2019a

systemcomposer.arch.Connector

Class that represents connector between ports

Description

The `Connector` class represents a connector between ports. This class is derived from `systemcomposer.arch.Element`.

Creation

Create a connector.

```
connector = connect(architecture,outports,inports)
```

Properties

`Parent` — Handle to parent architecture that owns connector
architecture object

Handle to 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 an 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 an input port, specified as a `systemcomposer.arch.ArchitecturePort` or `systemcomposer.arch.ComponentPort` object.

`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 element and through all operations that preserve the `UUID`.

Data Types: `char`

Model — Parent System Composer model
model object

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

SimulinkHandle — Simulink handle
numeric value

Simulink handle for connector, specified as a numeric value. This property is necessary for several Simulink related workflows and for using Simulink Requirement APIs.

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

Data Types: `double`

SimulinkModelHandle — Simulink handle to parent System Composer model
numeric value

Simulink handle to parent model of connector, specified as a numeric value. This property is necessary for several Simulink related workflows and for using Simulink Requirement APIs.

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

Data Types: `double`

Object Functions

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
getSourceElement	Gets signal interface elements selected on source port for connection
getDestinationElement	Gets signal interface elements selected on destination port for connection
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

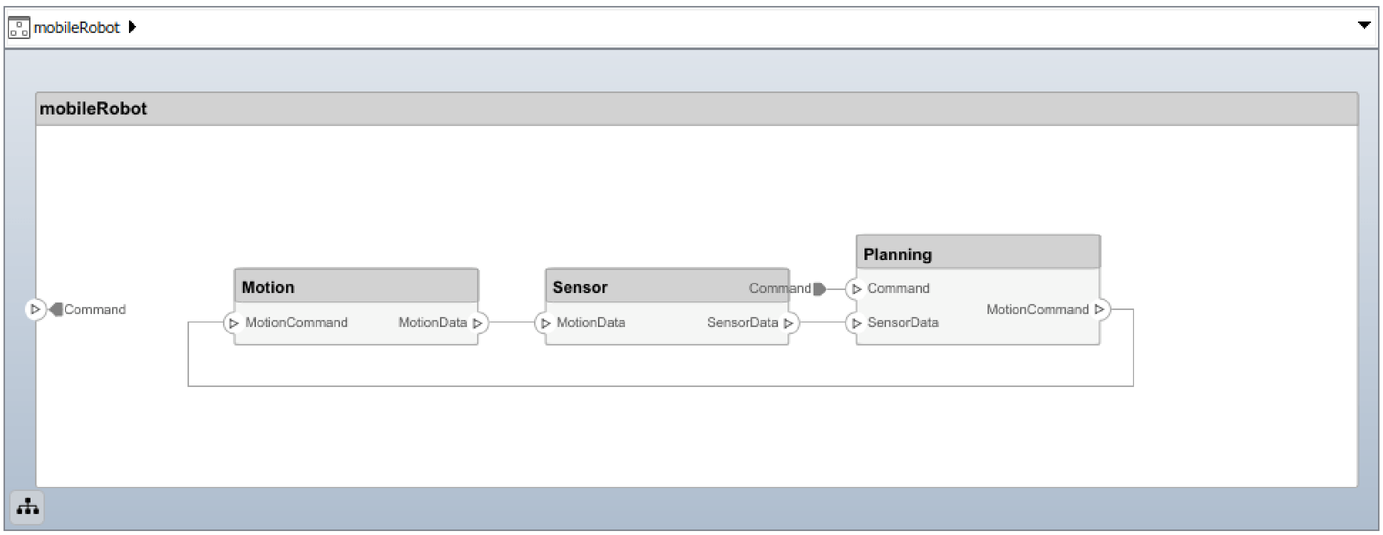
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

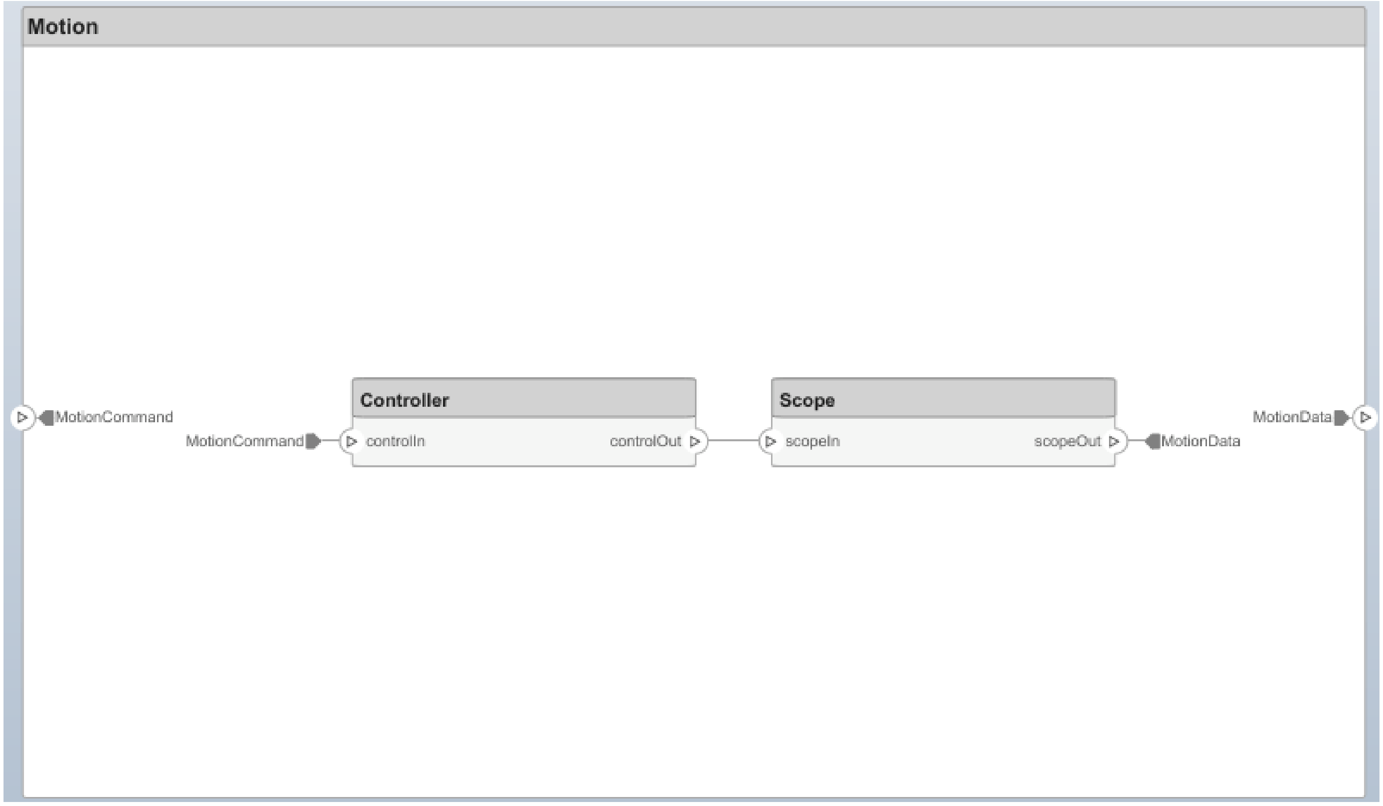
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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

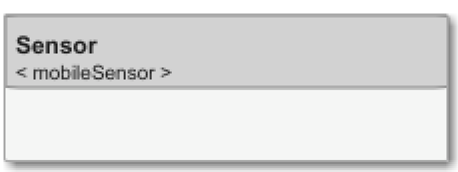


Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
```

```
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

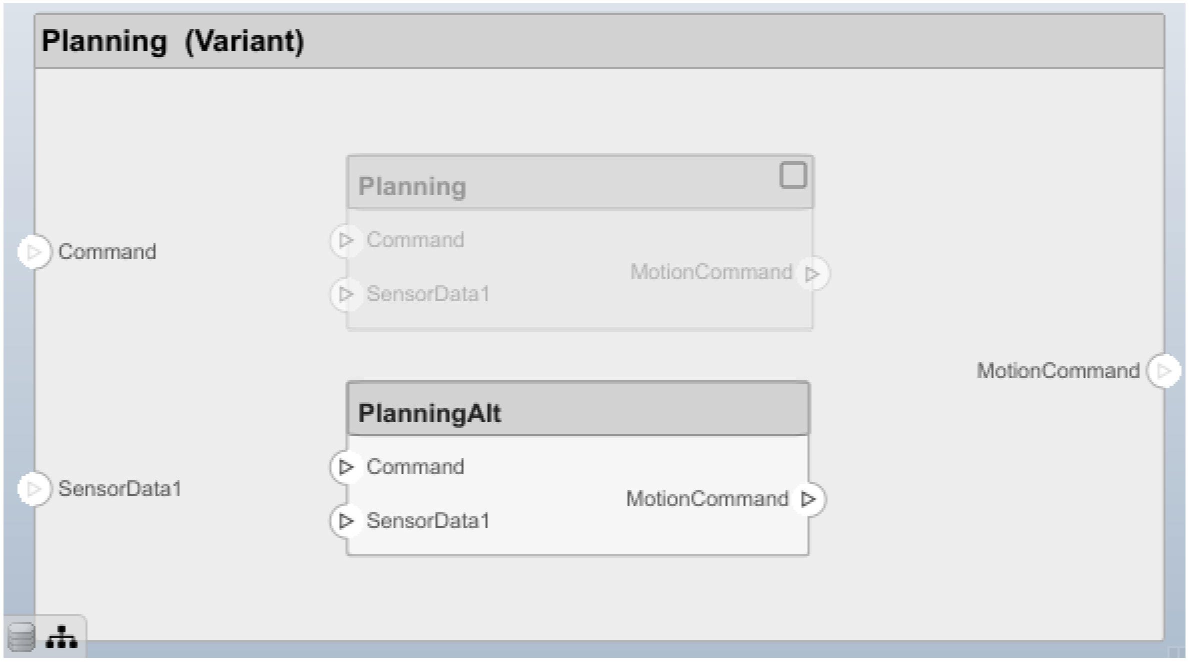
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 | `connect` | `systemcomposer.arch.Element`

Topics

“Create an Architecture Model”

Introduced in R2019a

systemcomposer.arch.Element

Base class of all model elements

Description

The `Element` class is the base class for all System Composer model elements — architecture, component, port, and connector.

- `systemcomposer.arch.Architecture`
- `systemcomposer.arch.Component`
- `systemcomposer.arch.VariantComponent`
- `systemcomposer.arch.BaseComponent`
- `systemcomposer.arch.ComponentPort`
- `systemcomposer.arch.ArchitecturePort`
- `systemcomposer.arch.BasePort`
- `systemcomposer.arch.Connector`

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 element and through all operations that preserve the `UUID`.

Data Types: `char`

`Model` — Parent System Composer model of element
model object

Parent model of element, specified as a `systemcomposer.arch.Model` object.

`SimulinkHandle` — Simulink handle
numeric value

Simulink handle for element, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

`SimulinkModelHandle` — Simulink handle to parent System Composer model of element
numeric value

Simulink handle to parent model of element, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

Object Functions

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
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

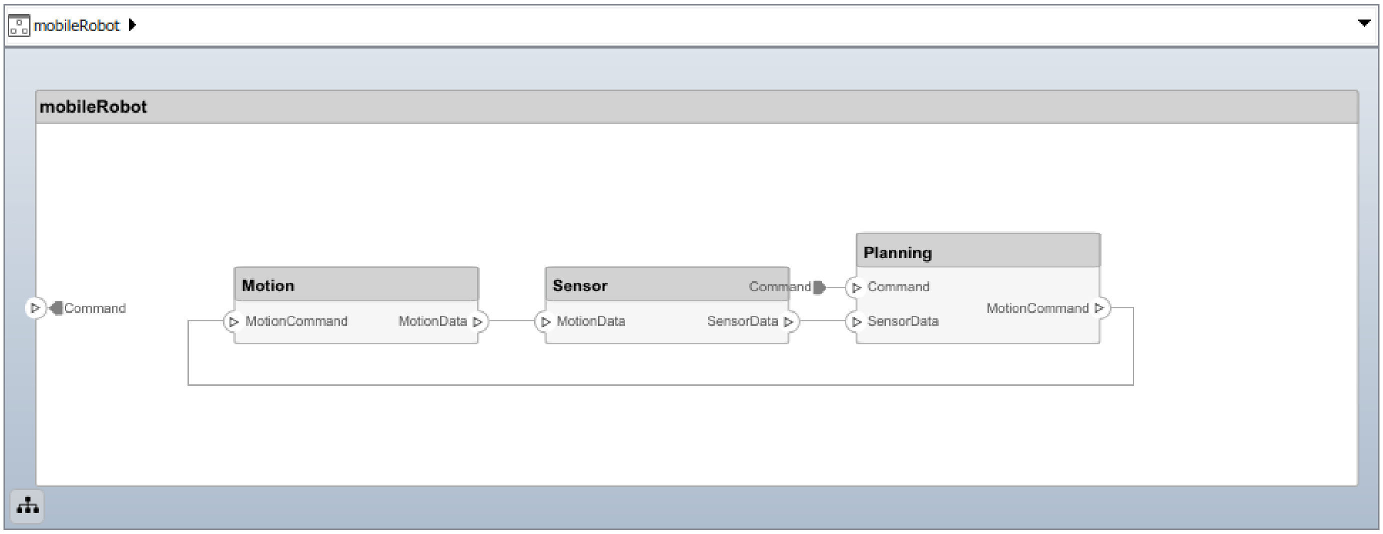
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

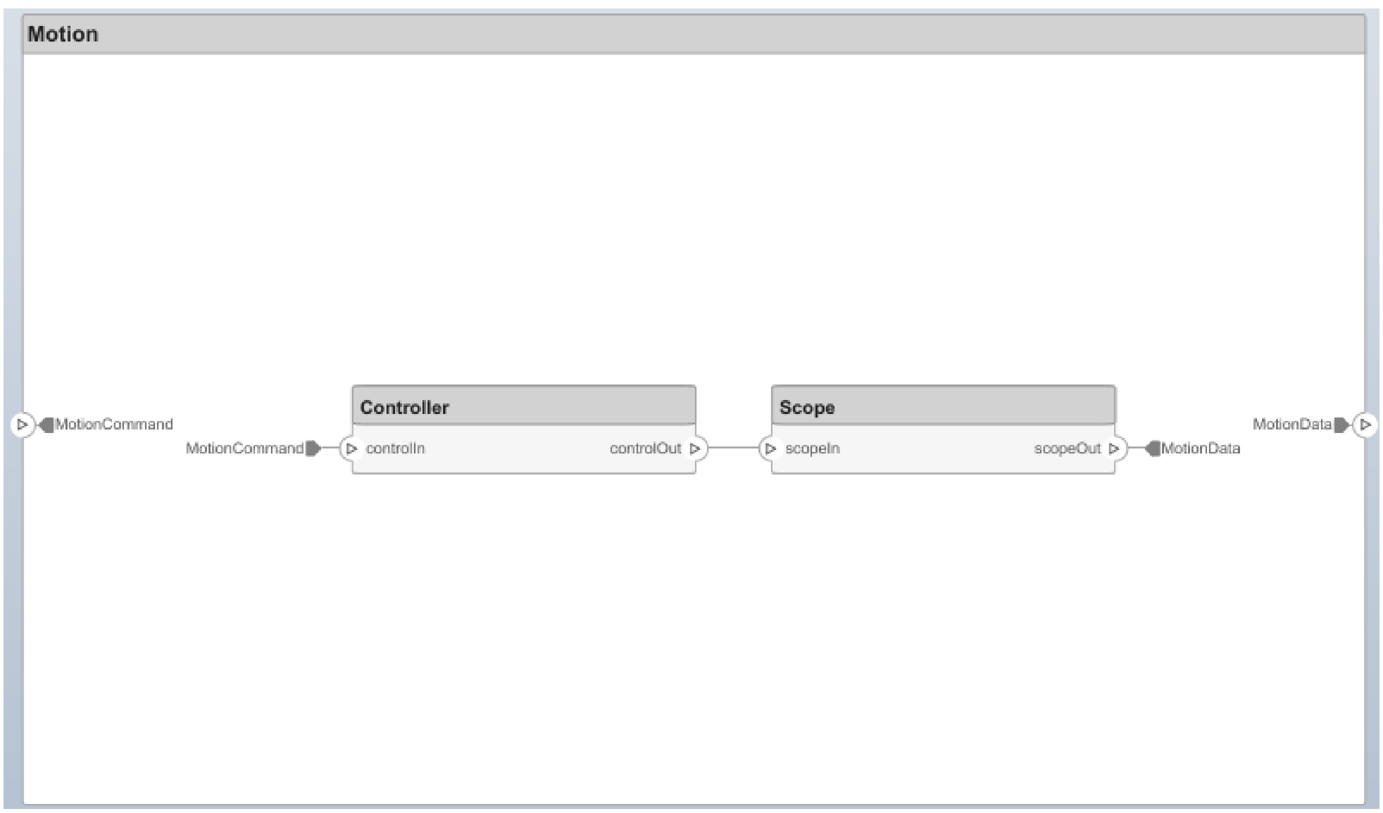
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



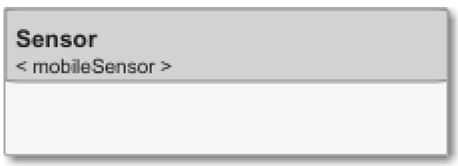
Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

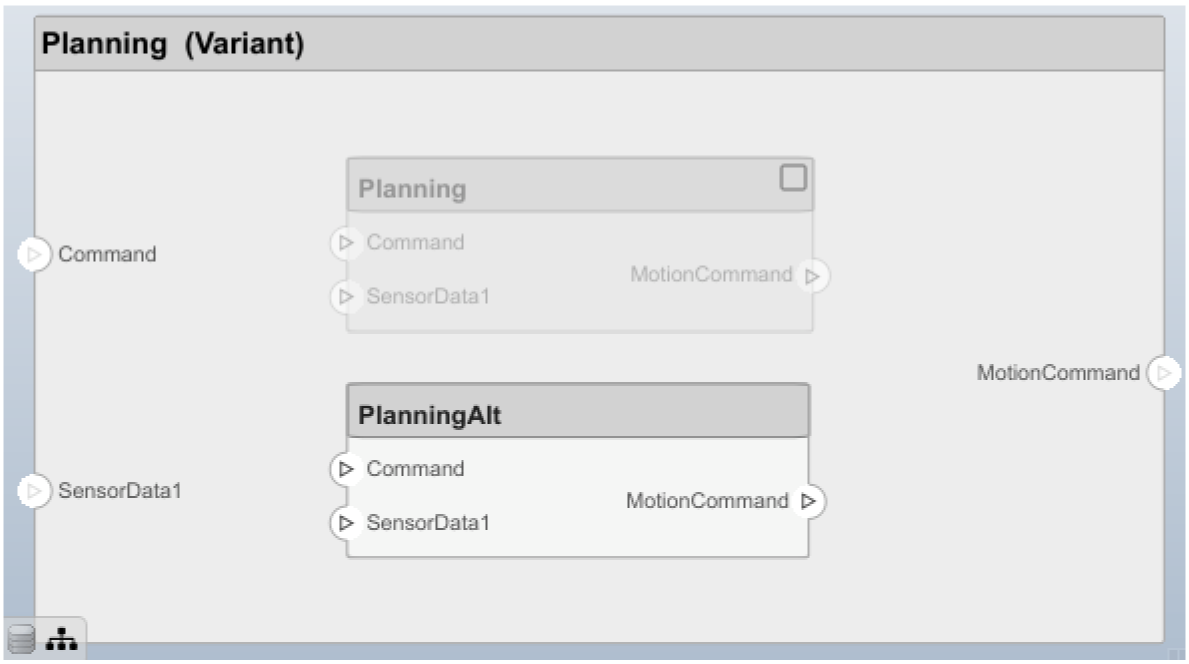
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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.Model

Class that represents System Composer model

Description

Use the `Model` class 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.

Example: `'NewModel'`

Data Types: `char`

`Architecture` — Root architecture of model
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 is a handle to the implicit dictionary

Views — Model views
array of view objects

Model views, specified as an array of `systemcomposer.view.View` objects.

Example: `objView = get(objModel,'Views')`

Object Functions

open	Open architecture model
close	Close model
save	Save architecture model or data dictionary
find	Find architecture model elements using query
lookup	Search for architecture element
openViews	Open architecture views editor
createView	Create architecture view
getView	Find architecture view
deleteView	Delete architecture view
applyProfile	Apply profile to model
removeProfile	Remove profile from model
saveToDictionary	Save interfaces to dictionary
linkDictionary	Link data dictionary to architecture model
unlinkDictionary	Unlink data dictionary from architecture model
renameProfile	Rename profile in model
iterate	Iterate over model elements

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

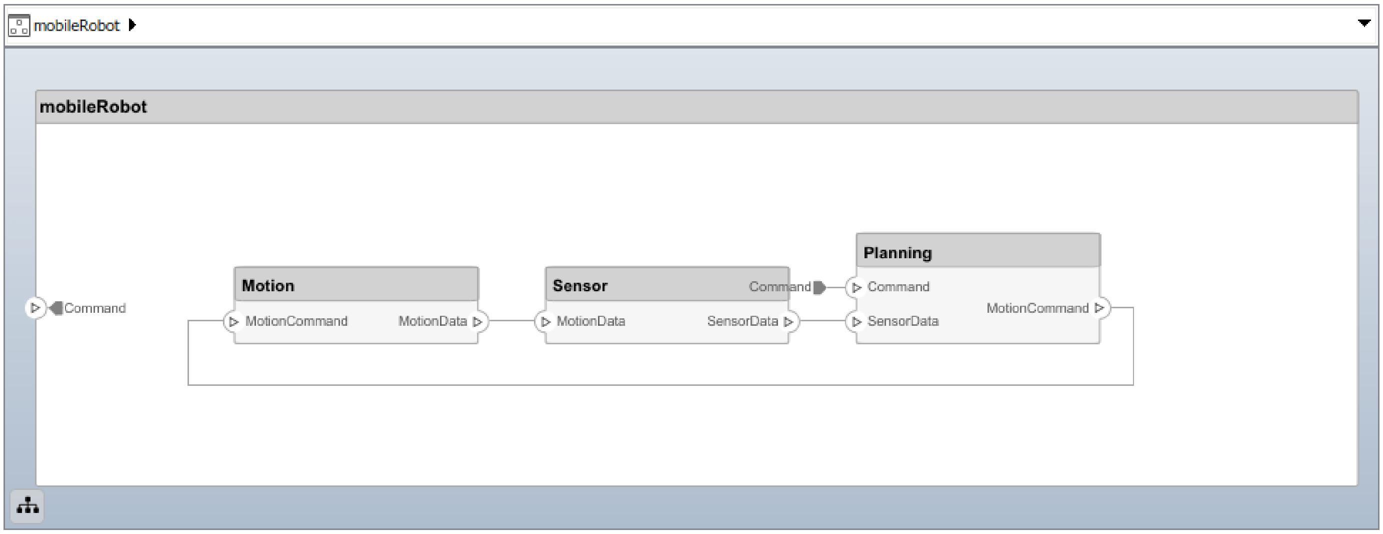
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressıng **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

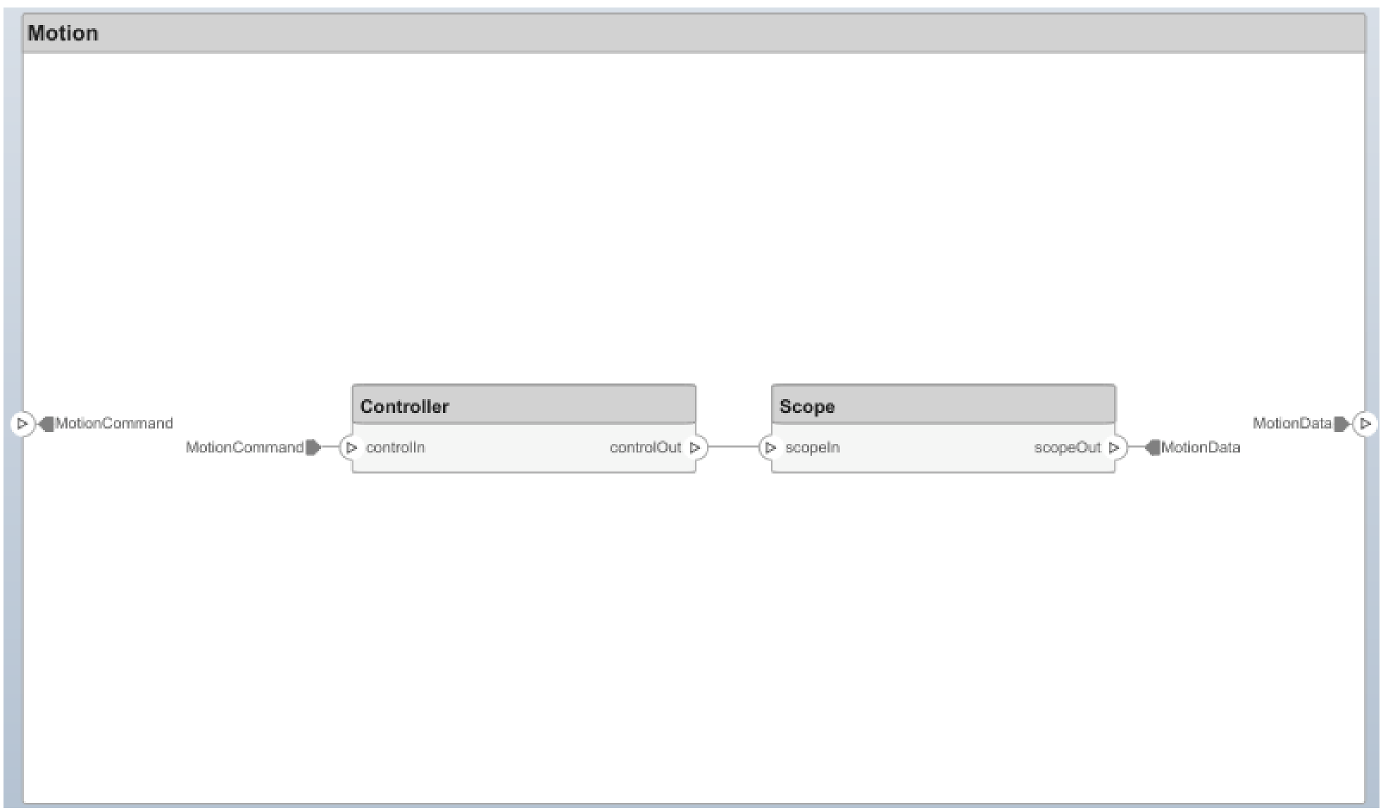
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



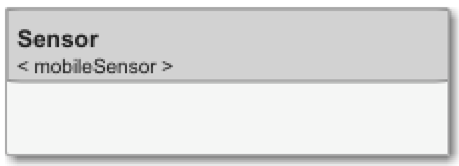
Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

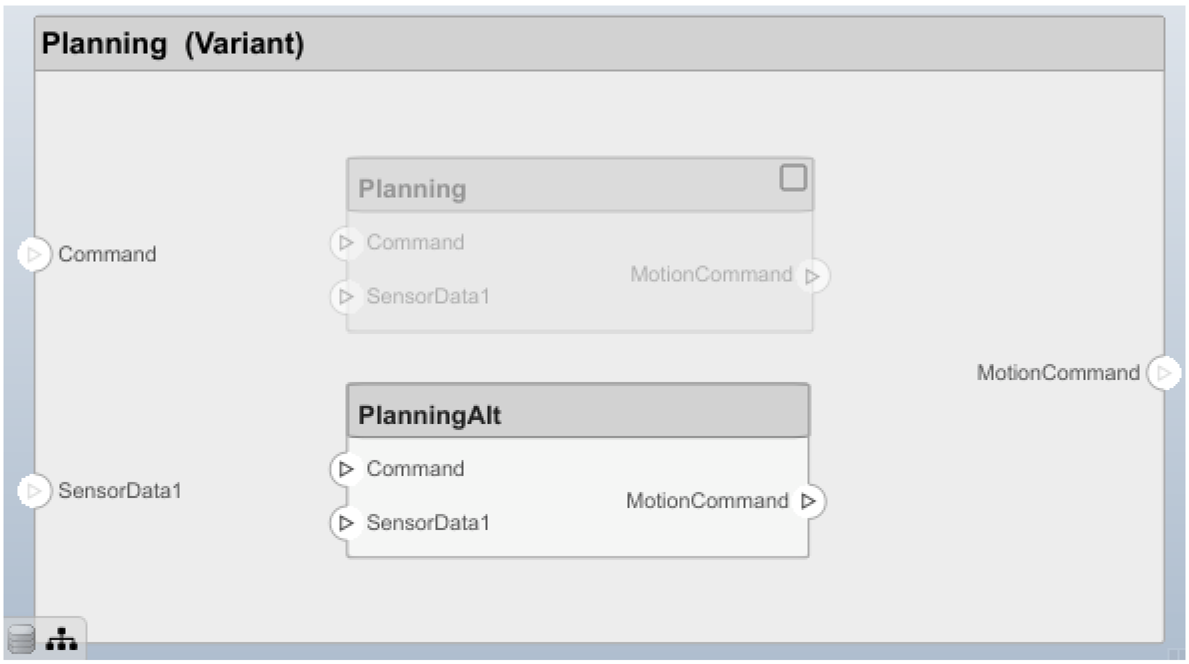
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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` | `exportModel` | `importModel` | `loadModel` | `openModel` | `saveAsModel`

Topics

"Create an Architecture Model"

Introduced in R2019a

systemcomposer.arch.VariantComponent

Class that represents variant component in System Composer model

Description

The `VariantComponent` class represents a variant component that allows you to create multiple design alternatives for a component. 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 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 element and through all operations that preserve the `UUID`.

Data Types: `char`

`Model` — Parent System Composer model
model object

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

`SimulinkHandle` — Simulink handle
numeric value

Simulink handle for component, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement APIs.

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

Data Types: `double`

`SimulinkModelHandle` — Simulink handle to parent System Composer model
numeric value

Simulink handle to parent model of component, specified as a numeric value. This property is necessary for several Simulink related work flows and for using Simulink Requirement 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
isReference	Find if component is reference to another model
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'i
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

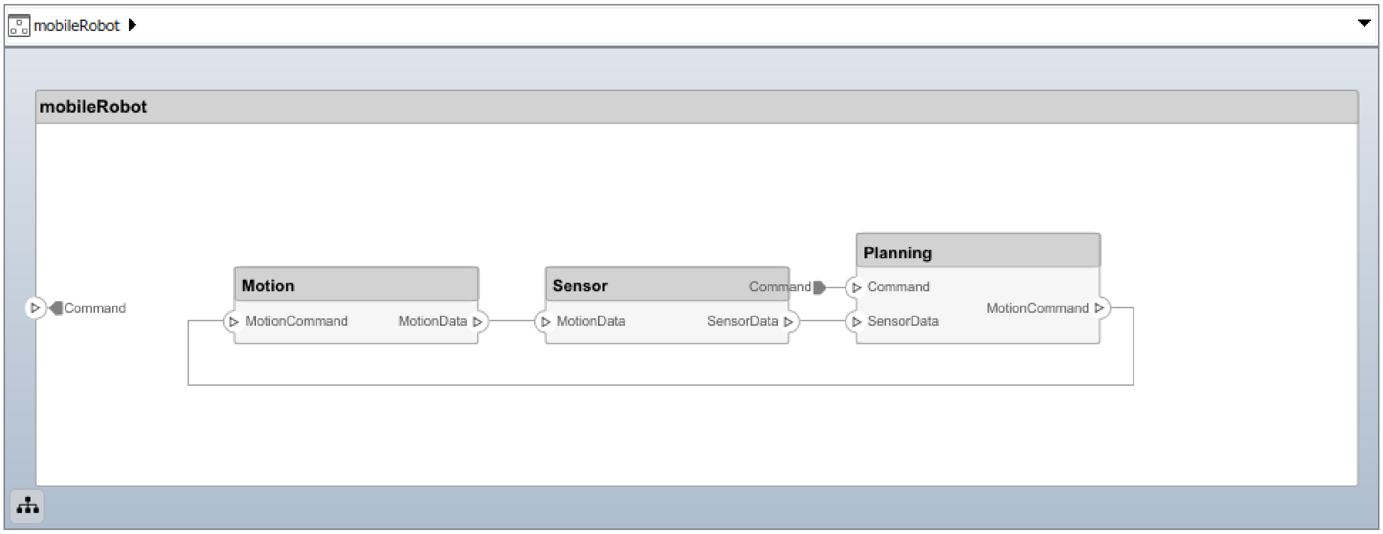
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer’s built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

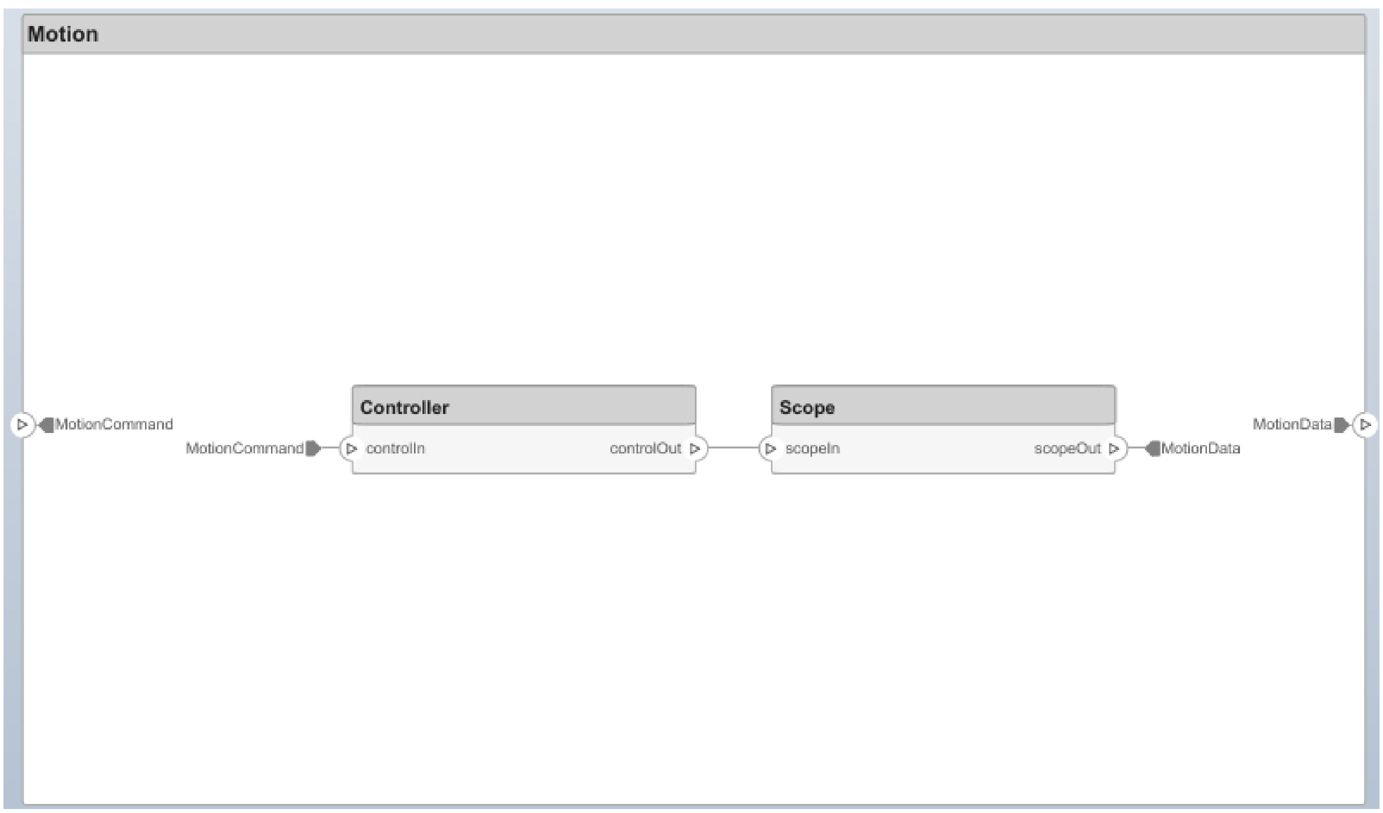
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



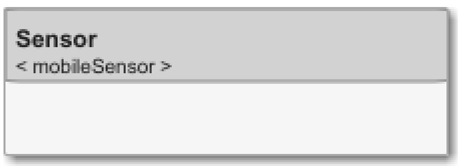
Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

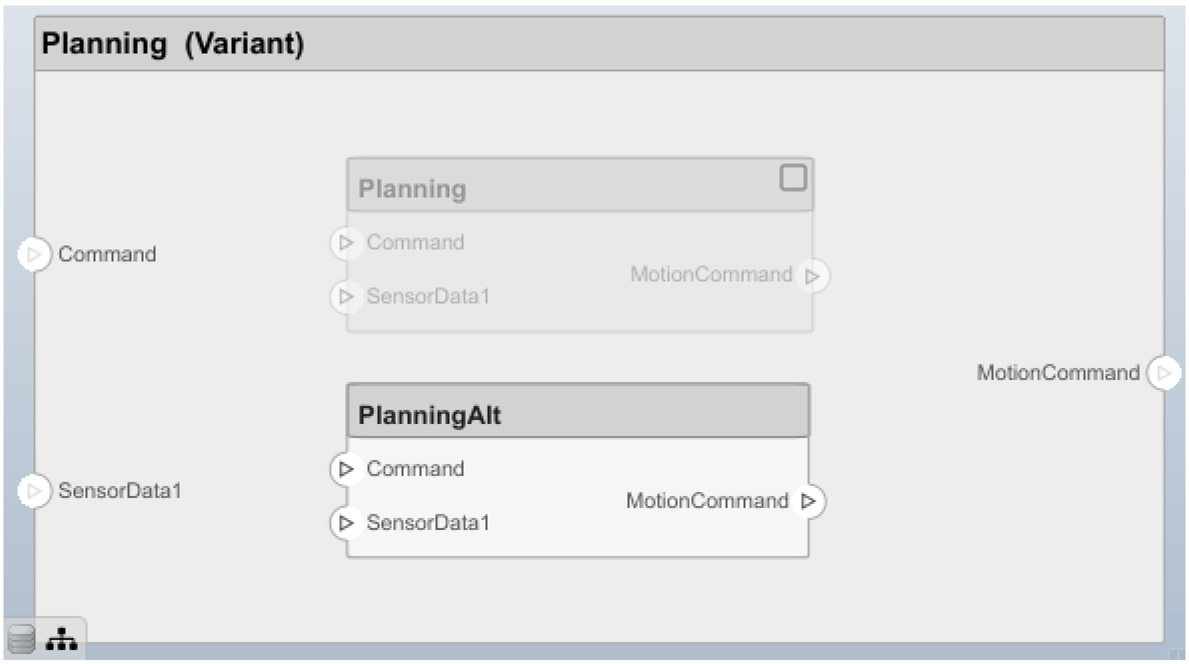
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 1-417

See Also
Variant Component

Topics
“Decompose and Reuse Components”

Introduced in R2019a

systemcomposer.interface.Dictionary

Class that represents interface dictionary of architecture model

Description

The `Dictionary` class represents the interface dictionary of an architecture model.

Creation

Create a dictionary.

```
dict_id = systemcomposer.createDictionary('newDictionary');
```

Properties

`Interfaces` — Interfaces defined in dictionary
array of signal interface objects

Interfaces defined in dictionary, specified as an array of `systemcomposer.interface.SignalInterface` 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 an interface dictionary, 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 interface dictionary and through all operations that preserve the `UUID`.

Data Types: `char`

Object Functions

applyProfile	Apply profile to model
removeProfile	Remove profile from model
addInterface	Create named interface in interface dictionary

getInterface	Get object for named interface in interface dictionary
getInterfaceNames	Get names of all interfaces in interface dictionary
removeInterface	Remove named interface from interface dictionary
save	Save architecture model or data dictionary
addReference	Add reference to dictionary
removeReference	Remove reference to dictionary
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

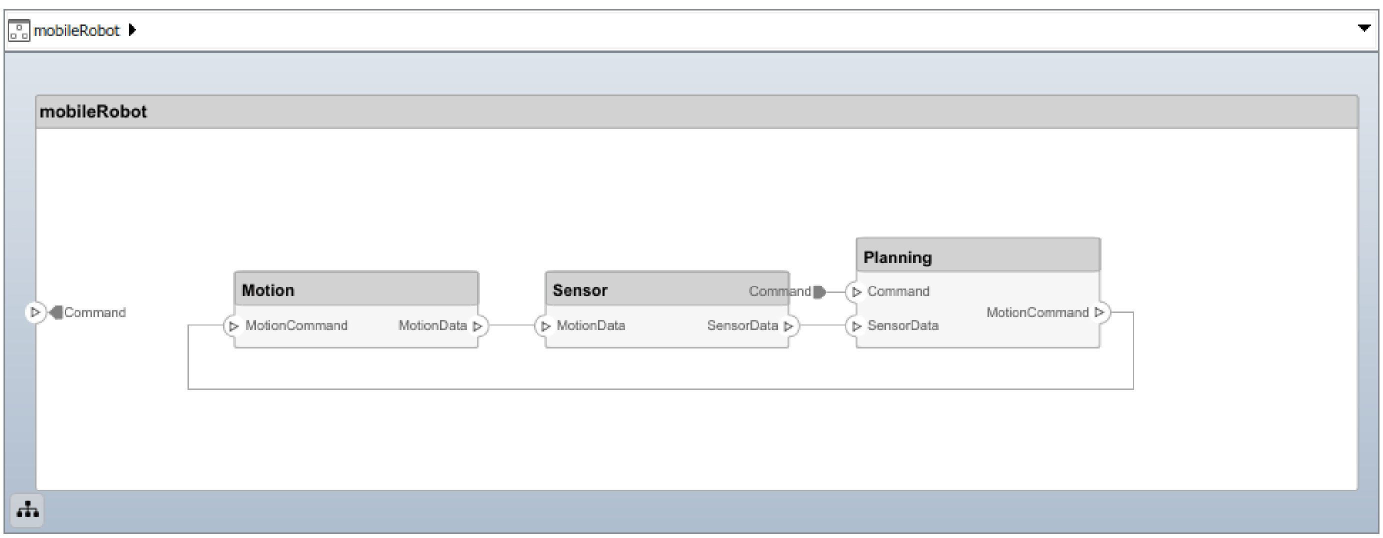
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

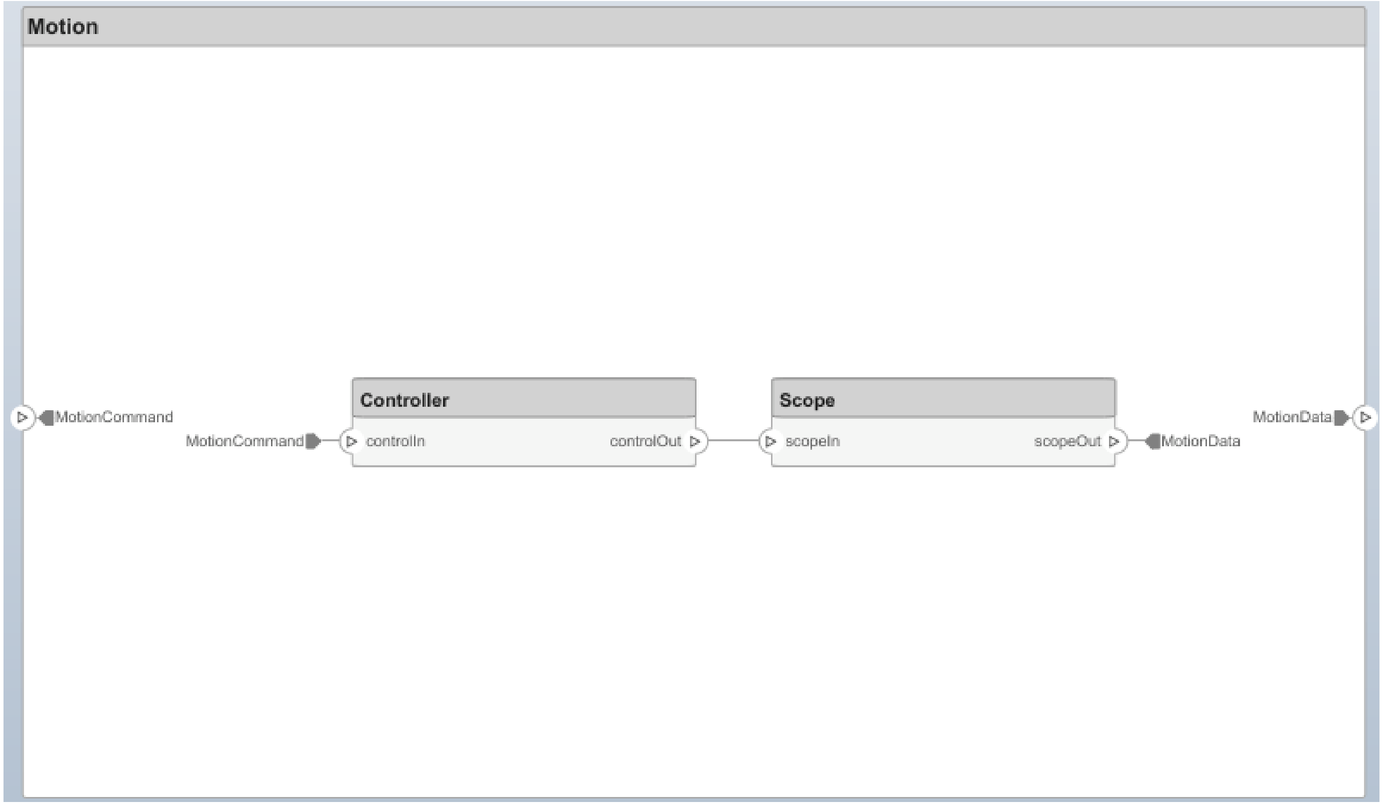
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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

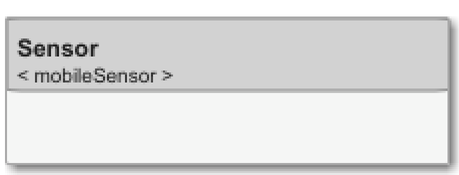


Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
```

```
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

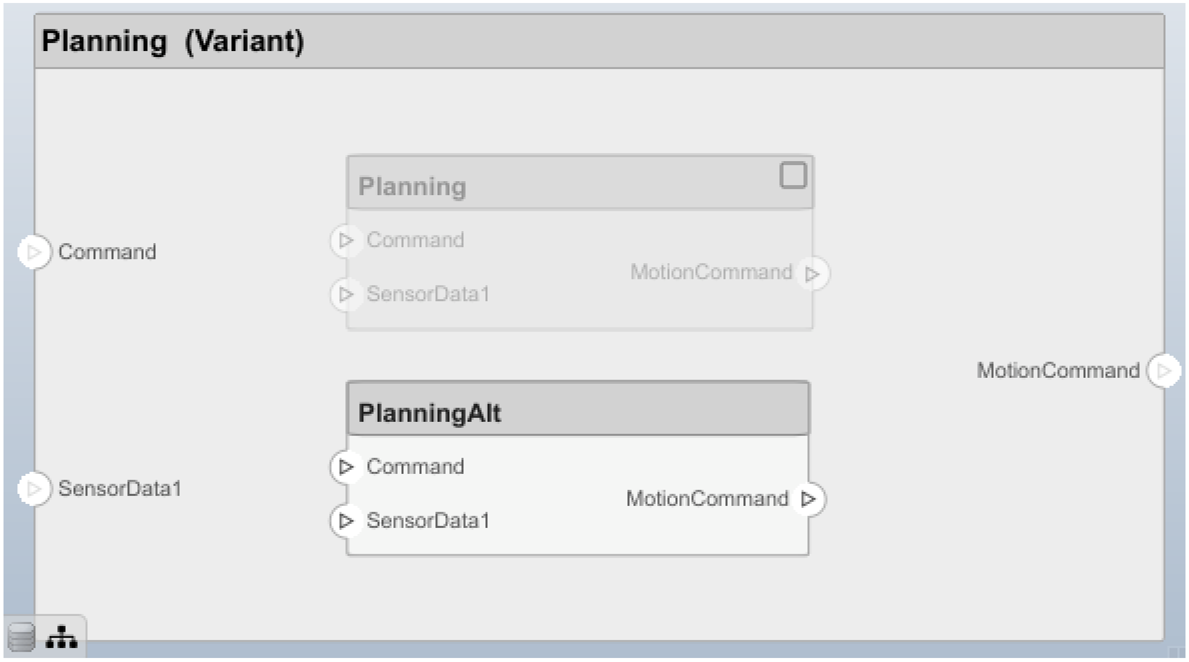
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"

Term	Definition	Application	More Information
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	“Assign Interfaces to Ports”
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• “Save, Link, and Delete Interfaces” • “Reference Data Dictionaries”
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	“Interface Adapter”

See Also

`createDictionary` | `openDictionary` | `saveToDictionary` | `systemcomposer.interface.SignalElement` | `systemcomposer.interface.SignalInterface`

Topics

“Define Interfaces”
“Assign Interfaces to Ports”
“Save, Link, and Delete Interfaces”
“Reference Data Dictionaries”

Introduced in R2019a

systemcomposer.interface.SignalElement

Class that represents element in signal interface

Description

The `SignalElement` class represents a single element in the signal interface.

Creation

Create a signal element.

```
element = addElement(interface,'NewElement')
```

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 `double`.

Data Types: `double`

`Maximum` — Maximum value for element
numeric

Maximum value for element, specified as a `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 an interface 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 interface element and through all operations that preserve the `UUID`.

Data Types: `char`

Object Functions

setName	Set name for signal interface element
setType	Set type for signal interface element
setDimensions	Set dimensions for signal interface element
setUnits	Set units for signal interface element
setComplexity	Set complexity for signal interface element
setMinimum	Set minimum for signal interface element
setMaximum	Set maximum for signal interface element
setDescription	Set description for signal interface element
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'i
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

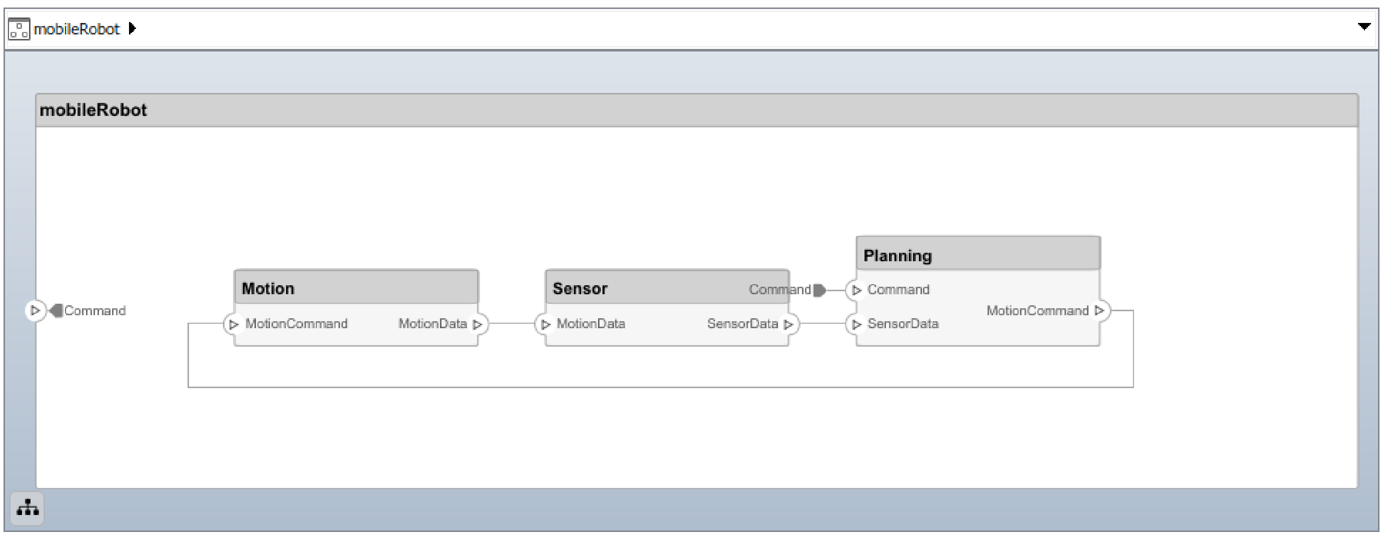
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
```

```
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

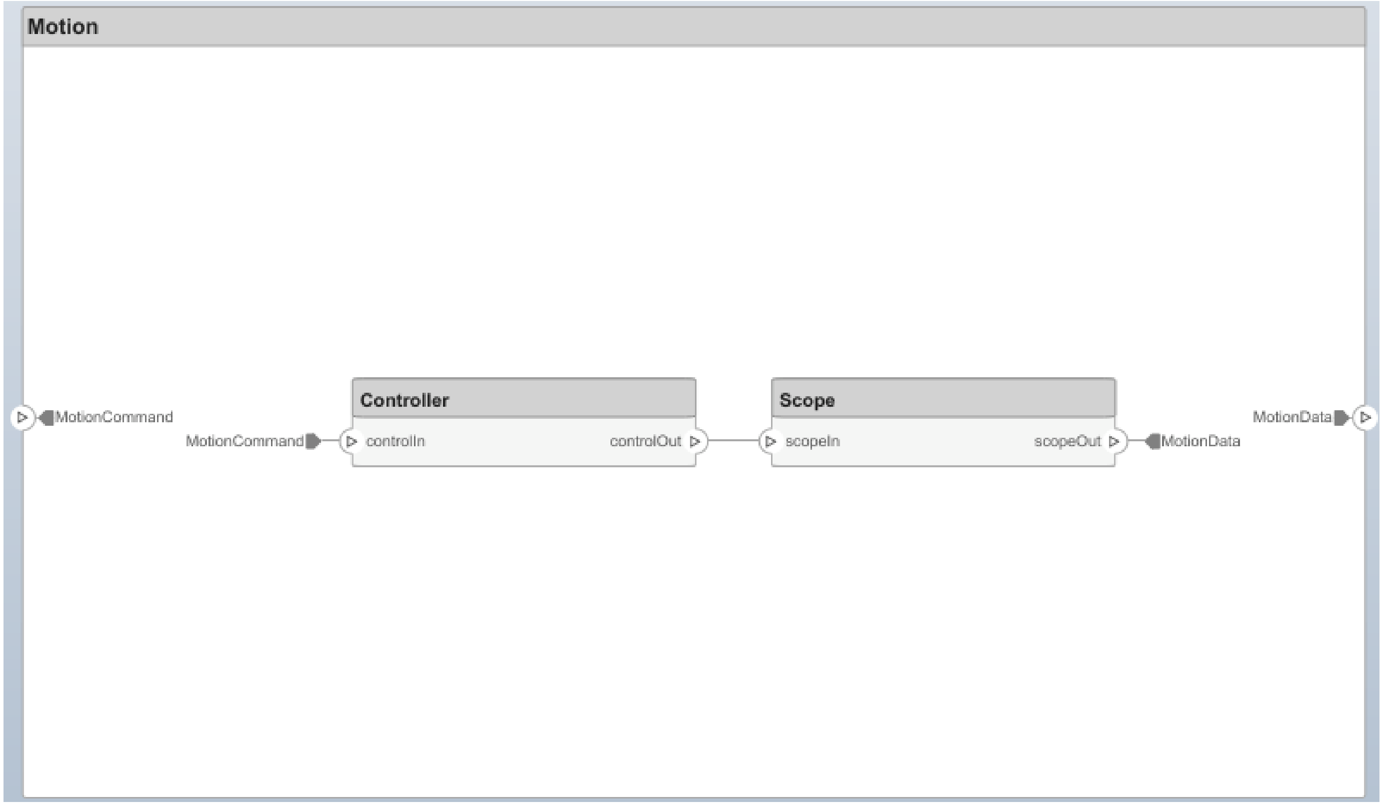
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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

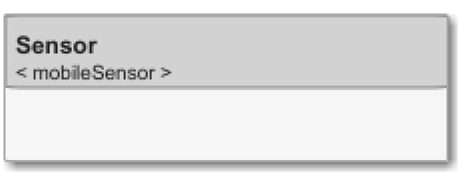


Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
```

```
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

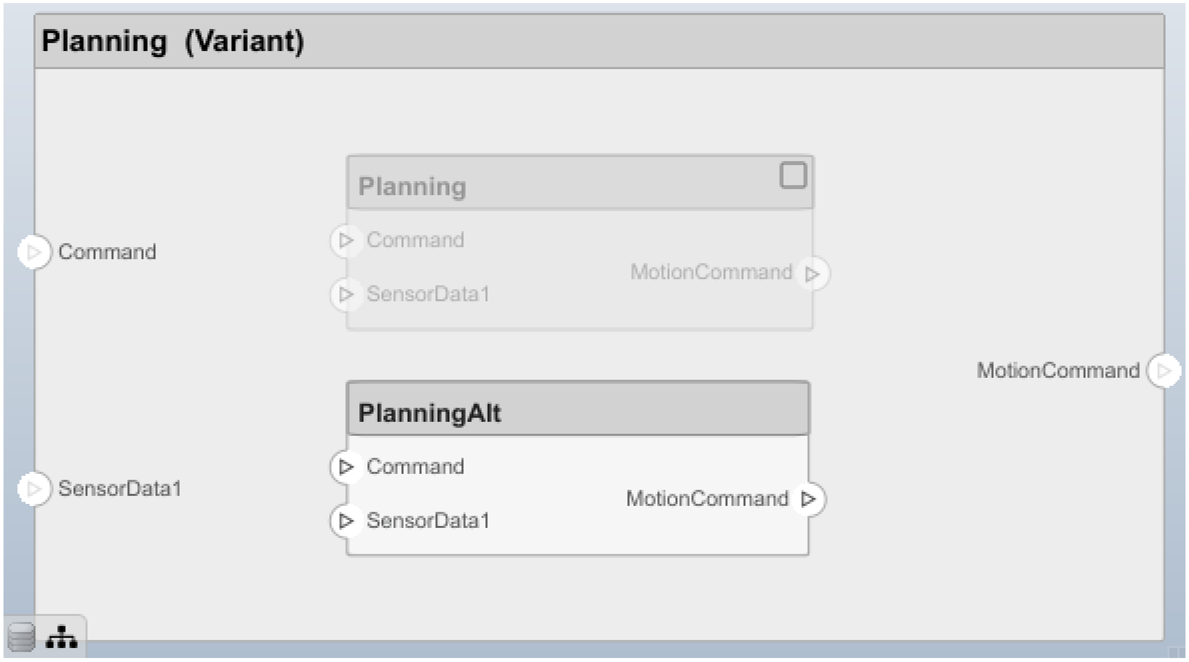
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	“Define Interfaces”

Term	Definition	Application	More Information
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"
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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addElement` | `addInterface` | `getElement` | `getInterface` | `getInterfaceNames` | `removeElement` | `removeInterface` | `systemcomposer.interface.Dictionary` | `systemcomposer.interface.SignalInterface`

Topics

"Define Interfaces"
"Assign Interfaces to Ports"
"Save, Link, and Delete Interfaces"

Introduced in R2019a

systemcomposer.interface.SignalInterface

Class that represents signal interface

Description

The `SignalInterface` class represents the structure of the signal interface at a given port.

Creation

Create an interface.

```
interface = addInterface(dictionary,'newInterface')
```

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`

`ExternalUID` — 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 System Composer model
model object

Parent model of signal interface, specified as a `systemcomposer.arch.Model` object.

Object Functions

addElement	Add signal interface element
getElement	Get object for signal interface element
removeElement	Remove signal interface element
applyStereotype	Apply stereotype to architecture model element
getStereotypes	Get stereotypes applied on element of architecture model
removeStereotype	Remove stereotype from model element
getProperty	Get property value corresponding to stereotype applied to element
getEvaluatedPropertyValue	Get evaluated value of property from component
setProperty	Set property value corresponding to stereotype applied to element
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
```

```
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'i
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

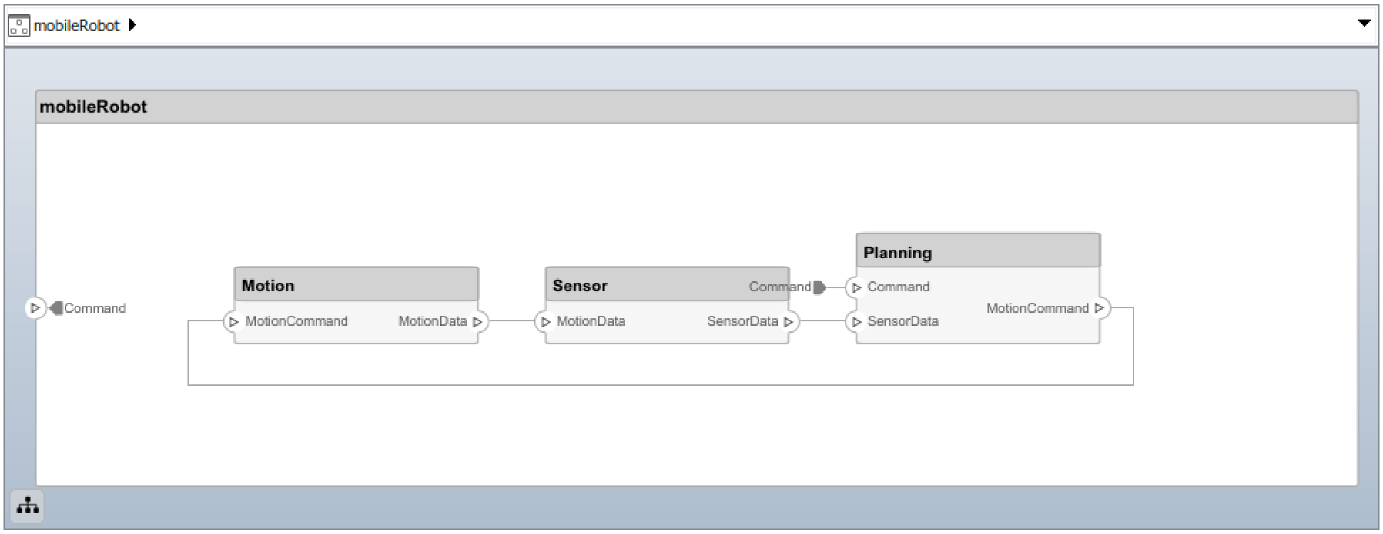
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

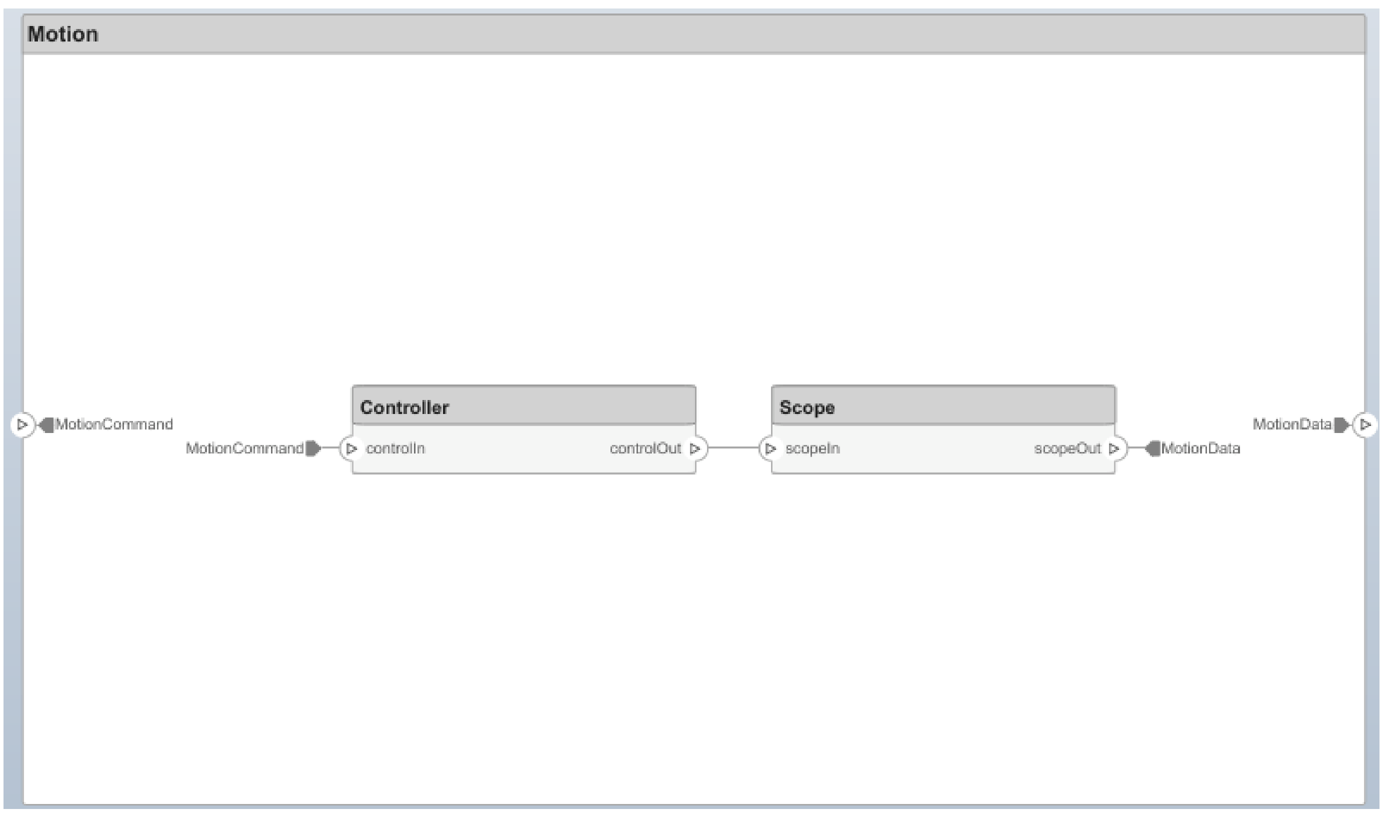
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



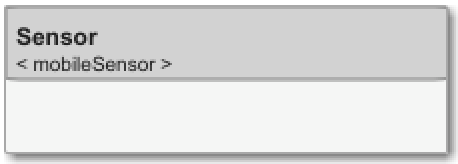
Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

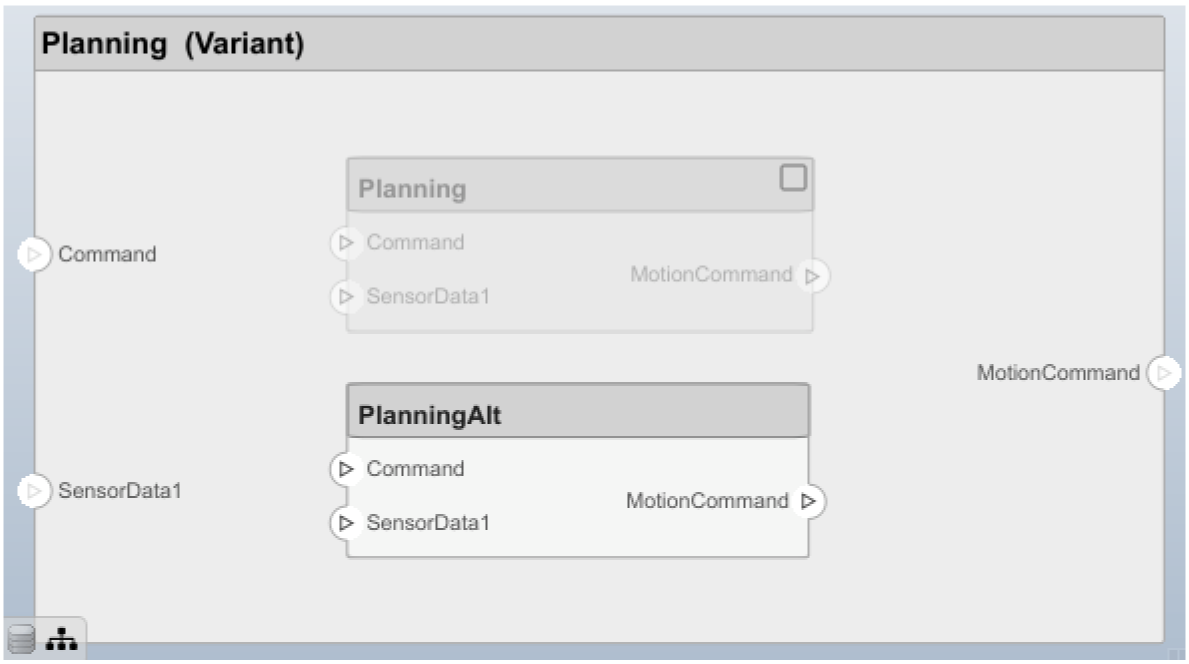
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

`addInterface` | `removeInterface` | `systemcomposer.interface.SignalElement`

Topics

"Define Interfaces"
"Assign Interfaces to Ports"
"Save, Link, and Delete Interfaces"

Introduced in R2019a

systemcomposer.io.ModelBuilder

Model builder for System Composer architecture models

Description

Build System Composer models using the model builder utility class. Build System Composer models with these sets of information: components and their position in architecture hierarchy, ports and their mappings to components, connections between the 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

Table containing the hierarchical information of components, type of component (for example, reference, variant, or adapter), stereotypes applied on component, and ability to set property values of component.

`Ports` — Ports information
table

Table containing the information about ports, their mappings to components and interfaces, as well as stereotypes applied on them.

`Connections` — Connections information
table

Table containing information about the connections between the ports defined in ports table also stereotypes applied on connections.

`Interfaces` — Interfaces information
table

Table containing the definitions of various interfaces and their elements.

Examples

Import System Composer Architecture Using Model Builder

This example shows how to 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 between the 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 e
  # 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 source excel 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 Model Builder Class

You can instantiate the model builder 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 external source file `DataDefinitions.xlsx`, we build the interface data model.

Create MATLAB® tables from source Excel 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

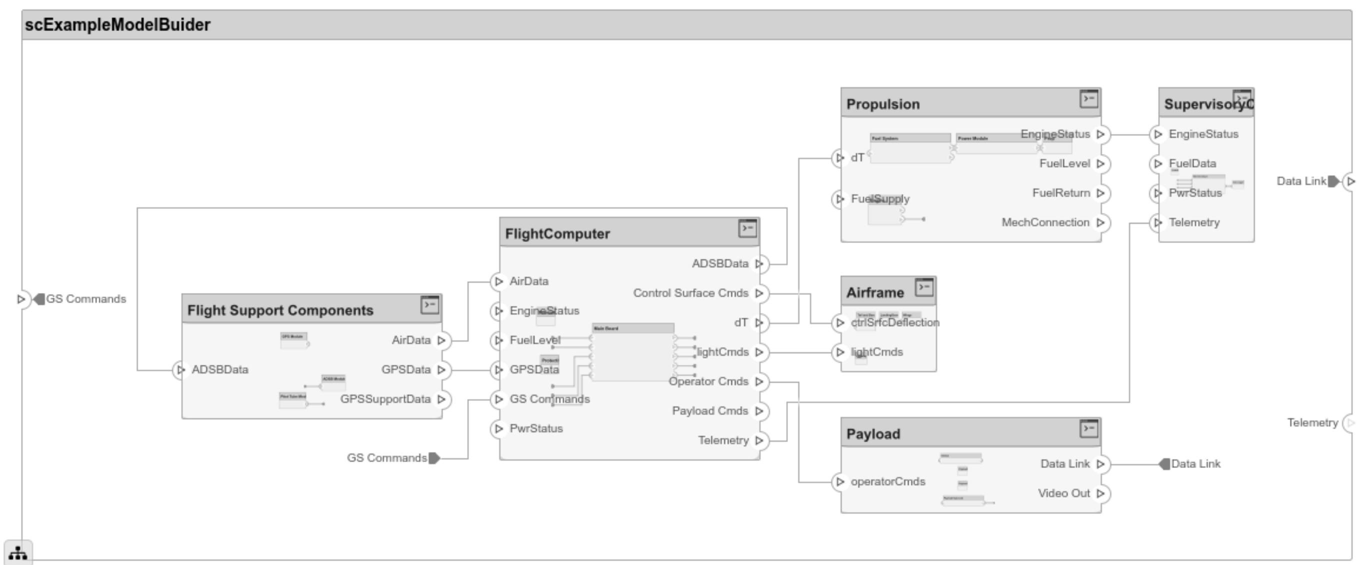
Architecture specifications are created by MATLAB tables from the source Excel file.

```
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. Builder build Method Imports Model from Populated Tables

```
[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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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
`addComponent(compName, ID, ParentID)`	Add component with name and ID as a child of component with ID as `ParentID`. In case of root, `ParentID` is 0.

`setComponentProperty(ID, varargin)`	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
	`builder.setComponentProperty(ID, 'StereotypeName',...`
	`'UAVComponent.PartDescriptor','ModelName',kind,'Manufacturer',doma`
	`ModelName` and `Manufacturer` are properties under stereotype `PartDescriptor`.

Ports	Description
`addPort(portName, direction, ID, compID)`	Add port with name and ID with direction (either Input or Output) to component with ID as `compID`.
`setPropertyOnPort(ID, varargin)`	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
`addConnection(connName, ID, sourcePortID,destPortID)`	Add connection with name and ID between ports with `sourcePortID` (direction: Output) and `destPortID` (direction: Input) defined in the ports table.
`setPropertyOnConnection(ID, varargin)`	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
`addInterface(interfaceName, ID)`	Add interface with name and ID to a data dictionary.
`addElementInInterface(elementName, ID, interfaceID, datatype, dimensions, units, complexity, Maximum, Minimum)`	Add element with name and ID under an interface with ID as `interfaceID`. Data types, dimensions, units, complexity, and maximum and minimum are properties of an element. These properties are specified as strings.
`addAnonymousInterface(ID, datatype, dimensions, units, complexity, Maximum, Minimum)`	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
`addInterfaceToPort(interfaceID, portID)`	Link an interface with ID specified as `InterfaceID` to a port with ID specified as `PortID`.

Models	Description
`build(modelName)`	Build model with model name passed as input.

Logging and Reporting	Description
`getImportErrorLog()`	Get `ErrorLogs` generated while importing the model . Called after the `build()` function
`getImportReport()`	Get a report of the import. Called after the `build()` function.

See Also

`exportModel` | `importModel`

Topics

"Import and Export Architecture Models"

Introduced in R2019b

systemcomposer.profile.Profile

Class that represents profile

Description

The `Profile` class represents architecture profiles.

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. 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 can contain spaces and special characters, but no new lines.

Data Types: `char`

`Description` — Description text for profile
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

createProfile	Create profile
addStereotype	Add stereotype to profile
removeStereotype	Remove stereotype from profile

getStereotype	Find stereotype in profile by name
getDefaultStereotype	Get default stereotype for profile
setDefaultStereotype	Set default stereotype for profile
find	Find profile by name
open	Open profile
load	Load profile from file
save	Save profile as file
close	Close profile
closeAll	Close all open profiles
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

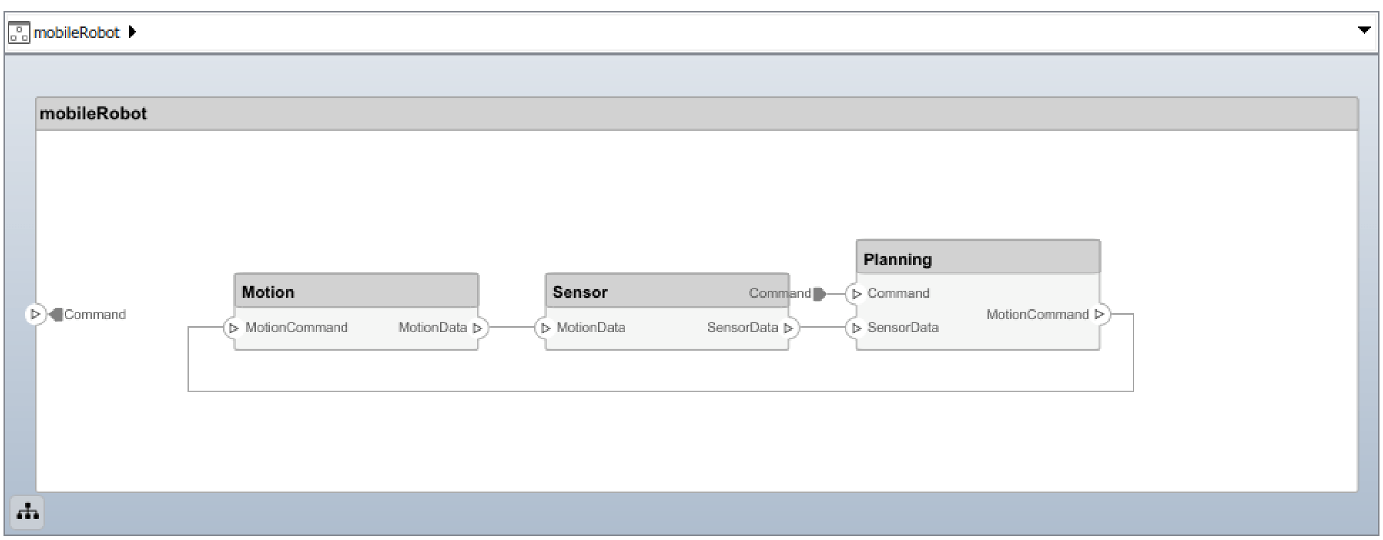
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

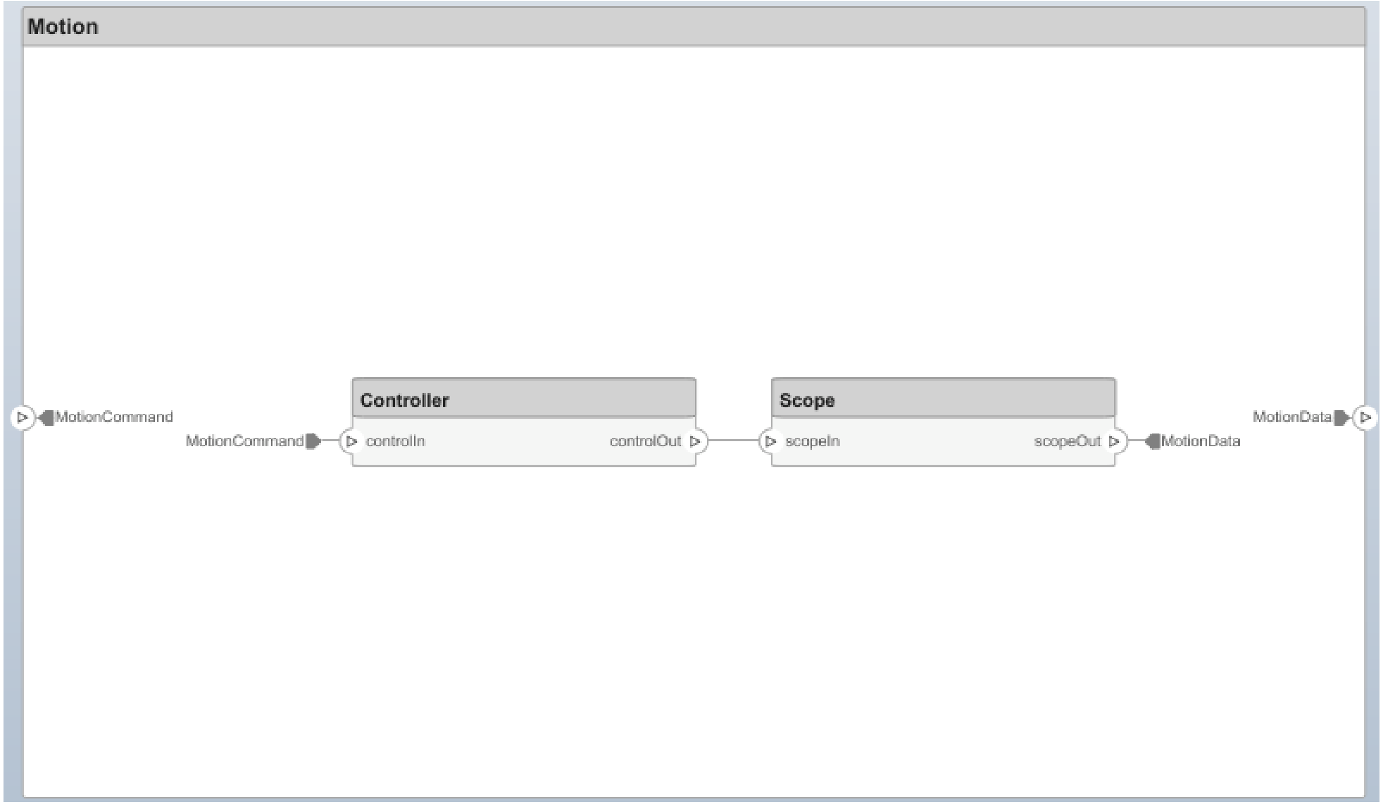
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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

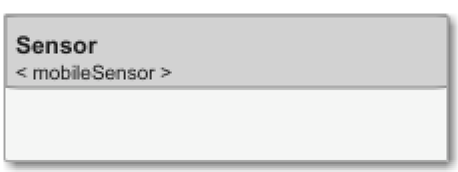


Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
```

```
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

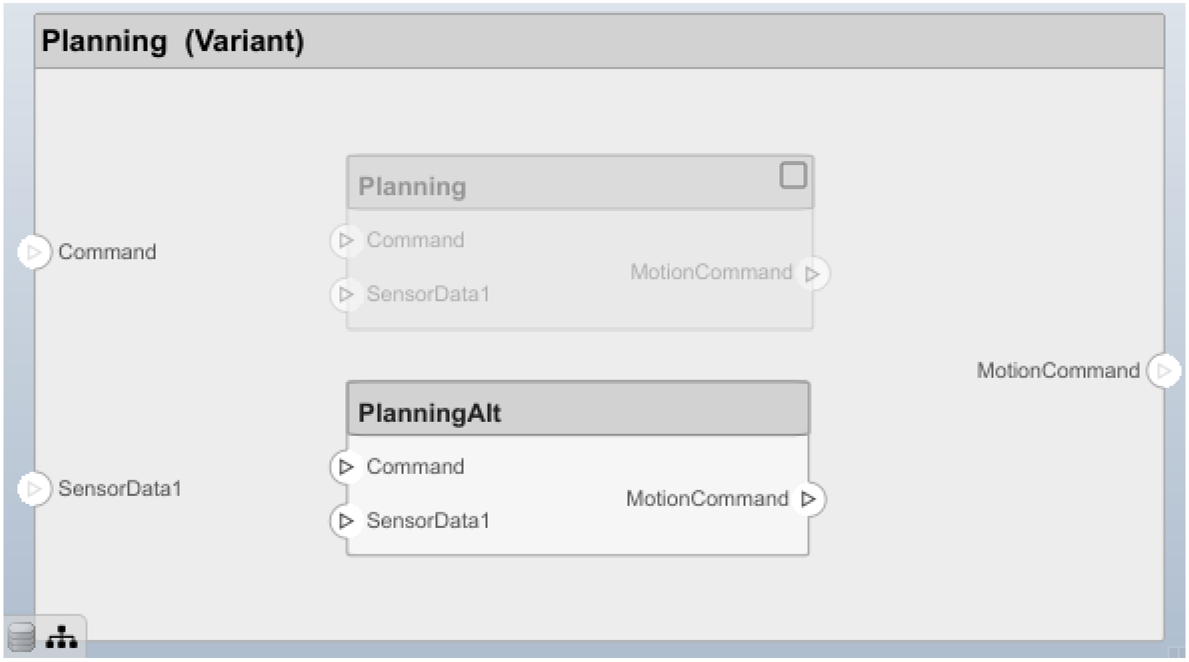
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`editor` | `loadProfile` | `systemcomposer.profile.Property` | `systemcomposer.profile.Stereotype`

Topics

"Define Profiles and Stereotypes"
"Use Stereotypes and Profiles"

Introduced in R2019a

systemcomposer.profile.Property

Class that represents property

Description

The `Property` class represents properties in a stereotype.

Creation

Add a property to a stereotype.

```
addProperty(stereotype,AttributeName,AttributeValue)
```

Properties

`Name` — Name of property
character vector

Name of property, specified as a character vector.

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 string values and units

Default value of property, specified as a string expression or an array of string value and string unit.

Data Types: `string`

`Stereotype` — Owning stereotype
stereotype object

Owning stereotype, specified as a `systemcomposer.profile.Stereotype` object.

Object Functions

destroy   Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'i
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

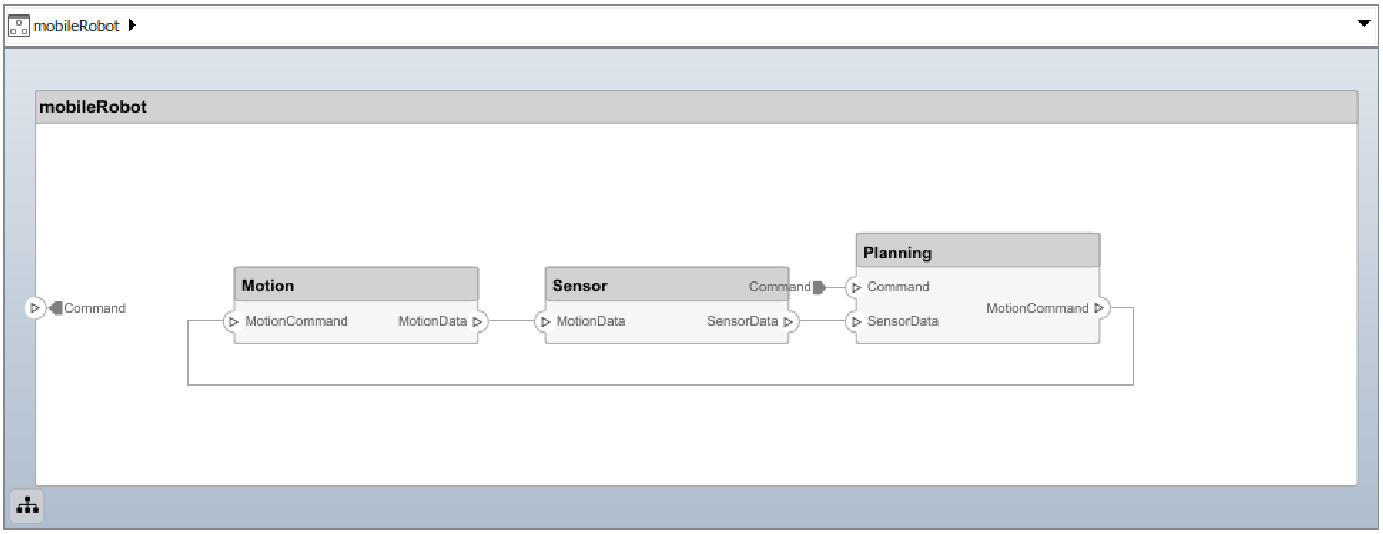
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressıng **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

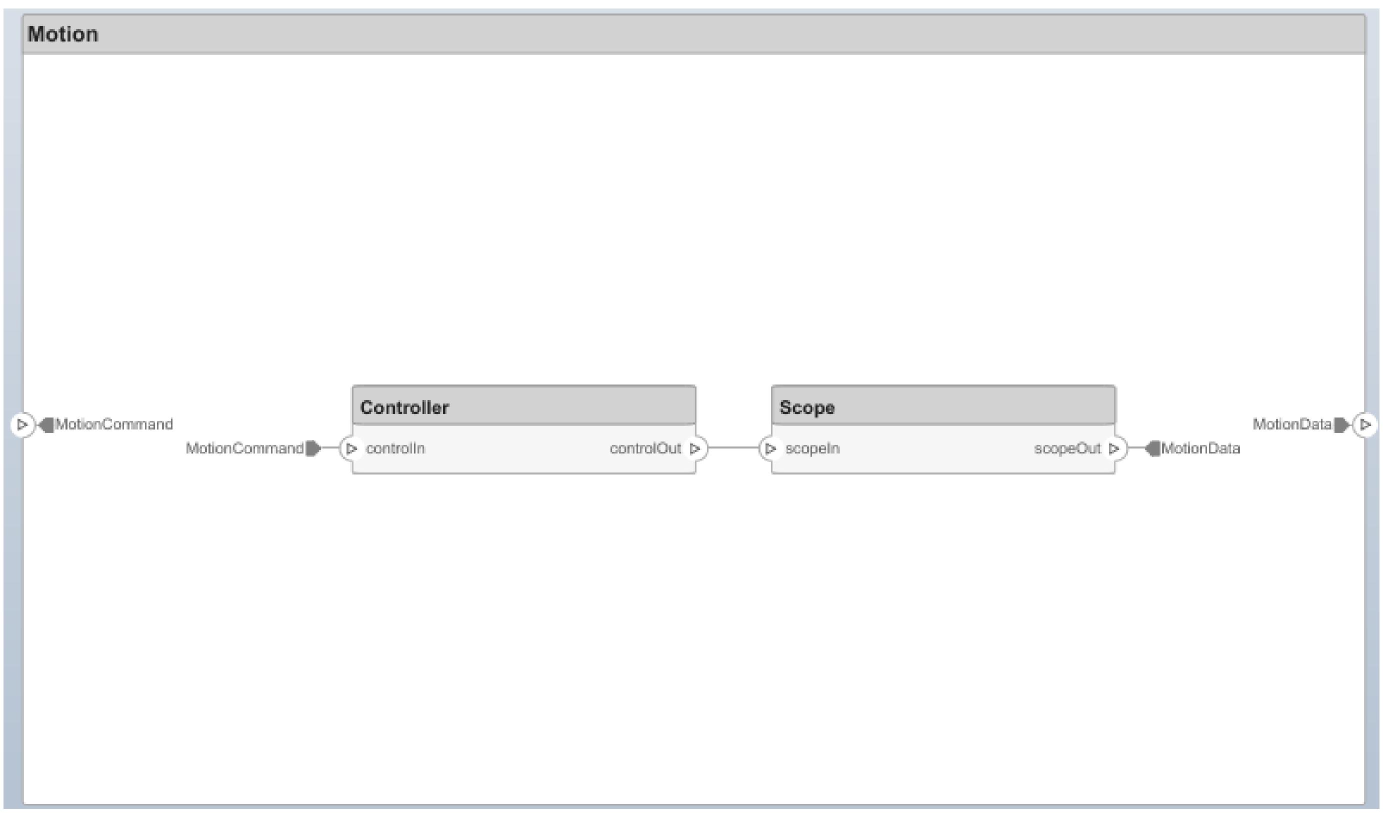
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



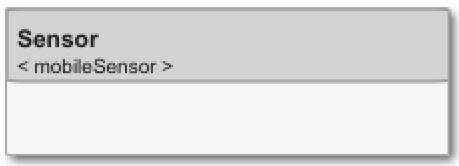
Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

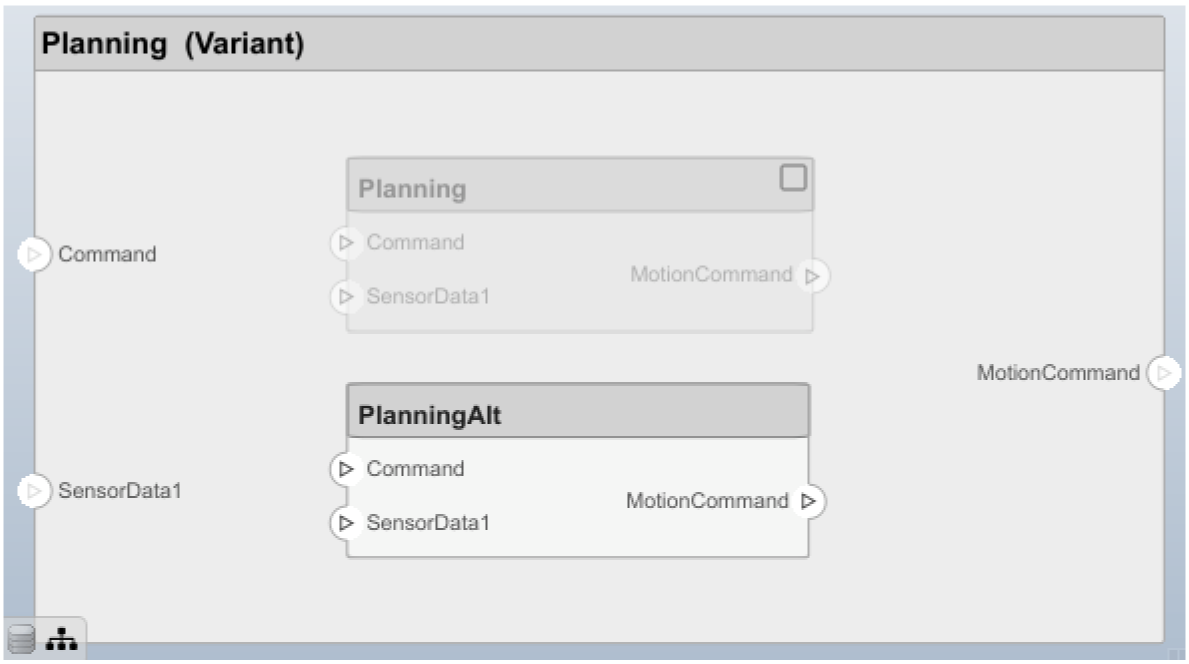
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	"Define Profiles and Stereotypes"
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`addProperty` | `removeProperty` | `systemcomposer.profile.Profile` | `systemcomposer.profile.Stereotype`

Topics

"Define Profiles and Stereotypes"
"Use Stereotypes and Profiles"

Introduced in R2019a

systemcomposer.profile.Stereotype

Class that represents stereotype

Description

The `Stereotype` class represents architecture stereotypes in a profile.

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.

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 of numeric 1 (`true`) or 0(`false`). If `true`, then 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
1x4 `uint32` row vector

Component header color, specified as a 1x4 `uint32` row vector in the form `Red Green Blue Alpha` The `Alpha` value determines the transparency.

Example: `206 232 246 255`

Data Types: `uint32`

`ConnectorLineColor` — Connector line color
1x4 `uint32` row vector

Connector line color, specified as a 1x4 `uint32` row vector in the form `Red Green Blue Alpha` The `Alpha` value determines the transparency.

Example: `206 232 246 255`

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 the stereotype
profile object

Stereotype from which stereotype inherits properties, specified as a `systemcomposer.profile.Profile` object.

`Properties` — Properties
cell array of character vectors

Properties contained in this 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

Owned properties contained in this stereotype, specified as a cell array of character vectors. The properties do not include properties inherited from the stereotype base hierarchy.

Data Types: `char`

Object Functions

addProperty	Define custom property for stereotype
removeProperty	Remove property from stereotype
find	Find stereotype by name
setDefaultComponentStereotype	Set default stereotype for components
setDefaultConnectorStereotype	Set default stereotype for connectors
setDefaultPortStereotype	Set default stereotype for ports
destroy	Remove model element

Examples

Build an Architecture Model from Command Line

This example shows how to build an architecture model using the System Composer™ API.

Prepare Workspace

Clear all profiles from the workspace.

```
systemcomposer.profile.Profile.closeAll;
```

Build a Model

To build a model, add a data dictionary with interfaces and interface elements, then add components, ports, and connections. After the model is built, you can create custom views to focus on a specific concern. You can also query the model to collect different model elements according to criteria you specify.

Add Components, Ports, and Connections

Create the model and extract its architecture.

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

Create data dictionary and add an interface. Link the interface to the model.

```
dictionary = systemcomposer.createDictionary('SensorInterfaces.sldd');
interface = addInterface(dictionary,'GPSInterface');
interface.addElement('Mass');
linkDictionary(model,'SensorInterfaces.sldd');
```

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

```
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface);

planningPorts = addPort(components(2).Architecture,{'Command','SensorData1','MotionCommand'},{'in
planningPorts(2).setInterface(interface);

motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Connect components with an interface rule. This rule connects ports on components that share the same interface.

```
c_sensorData = connect(arch,components(1),components(2),'Rule','interfaces');
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Add and Connect an Architecture Port

Add an architecture port on the architecture.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

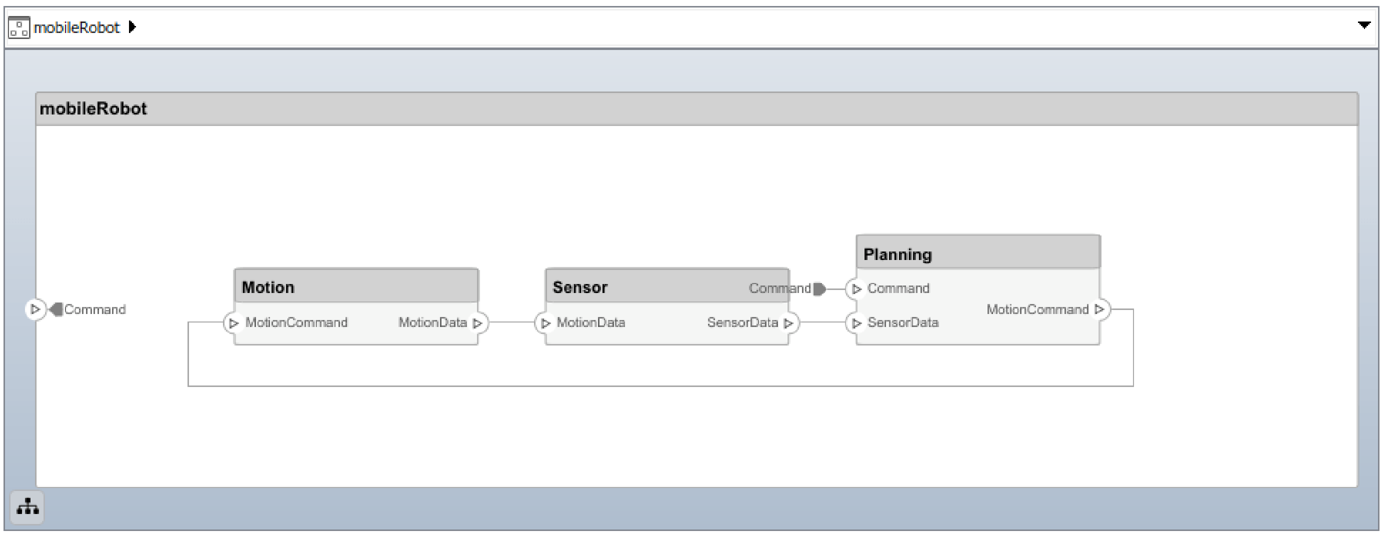
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Create and Apply Profile and 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 System Composer's built-in analysis capabilities, stereotypes can guide optimizations of your system for performance, cost, and reliability.

Create a 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. These types are dictated by design needs and are up to your discretion:

```
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 stereotypes. You can use properties to capture metadata for model elements and analyze non-functional 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 the Profile

```
save(profile);
```

Apply Profile to Model

Apply the profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'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(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'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 them 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.

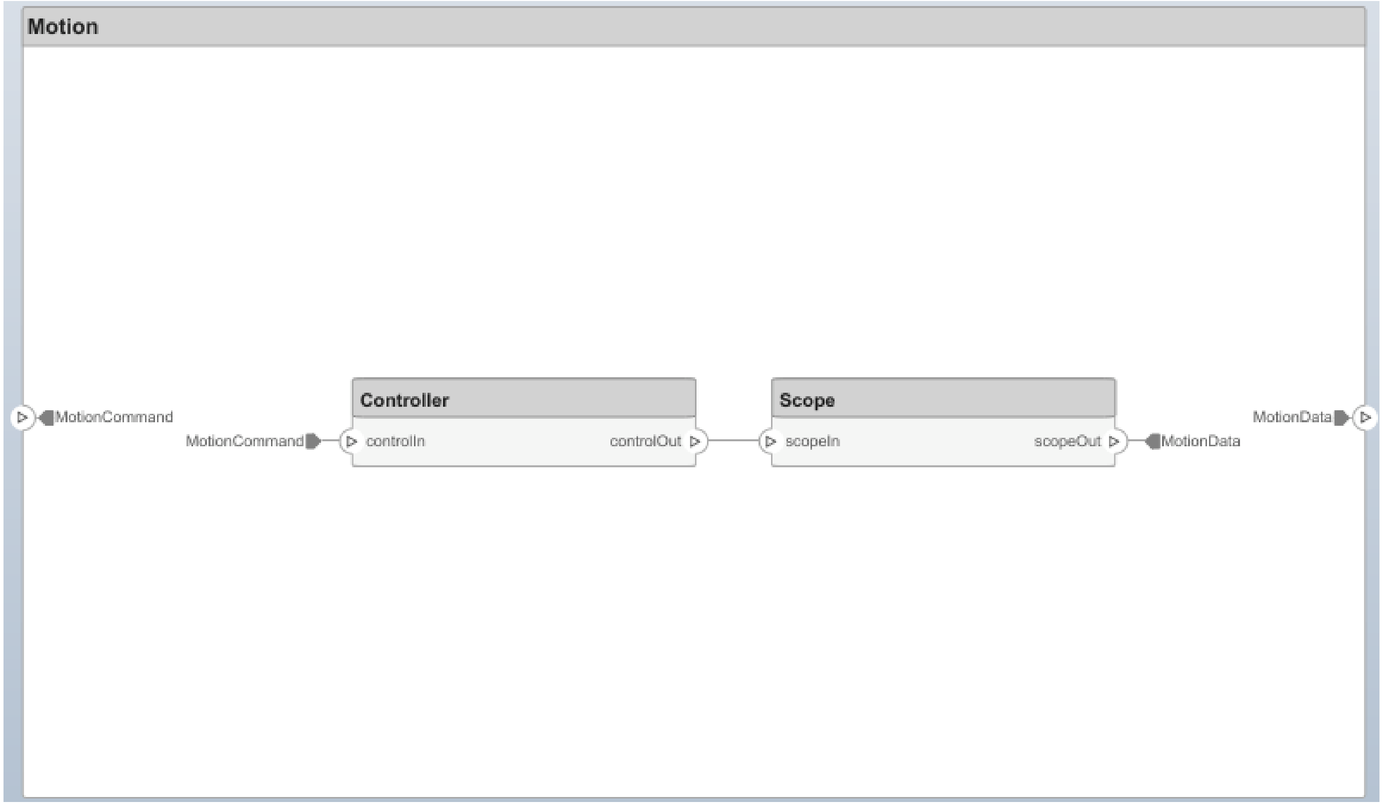
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

controllerPorts = addPort(motion(1).Architecture,{'controlIn','controlOut'},{'in','out'});
controllerCompPortIn = motion(1).getPort('controlIn');
controllerCompPortOut = motion(1).getPort('controlOut');

scopePorts = addPort(motion(2).Architecture,{'scopeIn','scopeOut'},{'in','out'});
scopeCompPortIn = motion(2).getPort('scopeIn');
scopeCompPortOut = motion(2).getPort('scopeOut');

c_planningController = connect(motionPorts(1),controllerCompPortIn);
c_planningScope = connect(scopeCompPortOut,motionPorts(2));
c_planningConnect = connect(controllerCompPortOut,scopeCompPortIn,'GeneralProfile.standardConn')
```

Save the model.

```
save(model)
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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

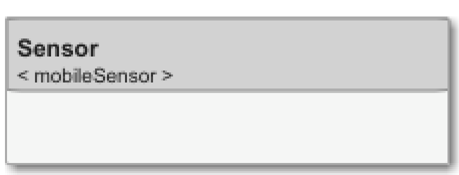


Create a Model Reference

Model references are useful to organize large models hierarchically and allow you to define architectures or behaviors once and reuse it. 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`.

```
newModel = systemcomposer.createModel('mobileSensor');
newArch = newModel.Architecture;
newComponents = addComponent(newArch,'ElectricSensor');
save(newModel);

linkToModel(components(1),'mobileSensor');
```



Apply a stereotype to the linked reference model's architecture and component.

```
referenceModel = get_param('mobileSensor','SystemComposerModel');
referenceModel.applyProfile('GeneralProfile');
```

```
referenceModel.Architecture.applyStereotype('GeneralProfile.softwareComponent');
batchApplyStereotype(referenceModel.Architecture,'Component','GeneralProfile.projectElement')
```

Add ports and connections to the reference component.

```
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
sensorPorts(2).setInterface(interface)
connect(arch,components(1),components(2),'Rule','interfaces');
connect(arch,components(3),components(1));
```

Save the models.

```
save(referenceModel)
save(model)
```

Make a Variant Component

You can convert the `Planning` component into 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 behaviorial designs programmatically in an architecture model to perform trade studies and analysis.

```
[variantComp,choice1] = makeVariant(components(2));
```

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`.

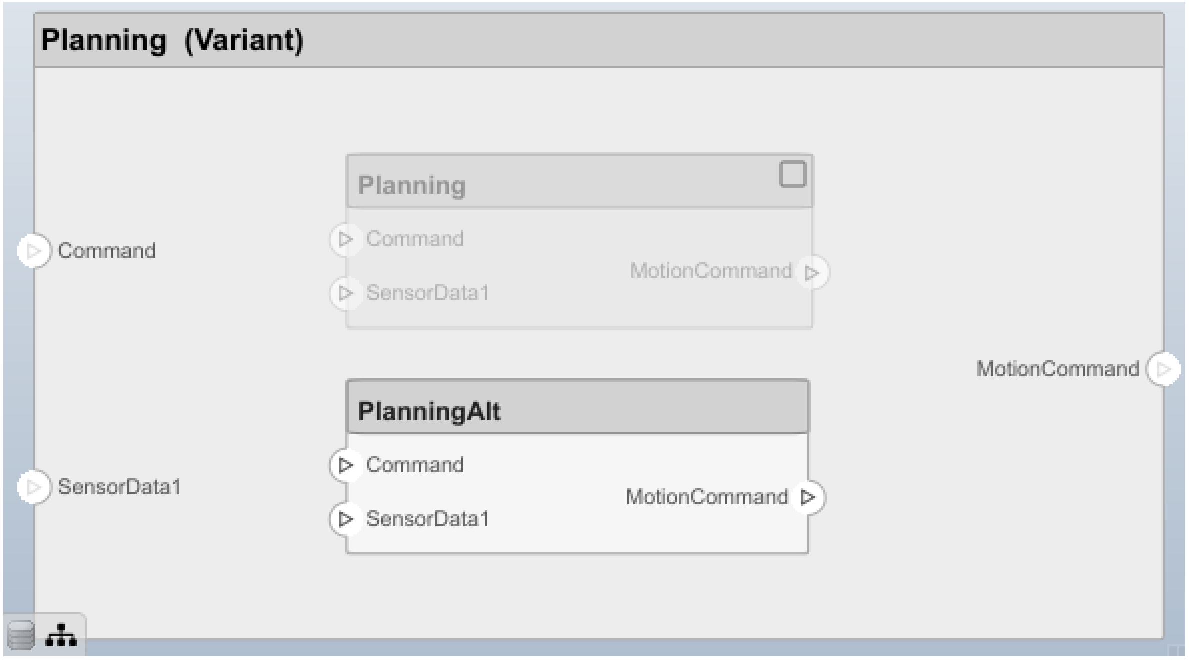
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

Make `PlanningAlt` the active variant.

```
setActiveChoice(variantComp,'PlanningAlt')
```

Arrange the layout by pressing **Ctrl+Shift+A** or using the following command:

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



Save the model.

```
save(model)
```

Clean Up

Uncomment the following code and run 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 the root level architecture, component architecture, connectors, ports, and interfaces of a model. Stereotypes provide model elements within the architecture a common set of property fields, such as mass, cost, and power.	“Define Profiles and Stereotypes”

Term	Definition	Application	More Information
profile	A profile is a package of stereotypes to create a self-consistent domain of model element types.	Apply profiles to a model through 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.	"Use Stereotypes and Profiles"
property	A property is a field in a stereotype. For each model 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.	"Set Properties"

See Also

`addStereotype` | `getStereotype` | `removeStereotype` | `systemcomposer.profile.Profile`

Topics

"Define Profiles and Stereotypes"
"Use Stereotypes and Profiles"

Introduced in R2019a

systemcomposer.query.Constraint

Class that represents query constraint

Description

The `Constraint` class is the base class for all System Composer query constraints.

Object Functions

AnyComponent	Create query to select all components in model
IsStereotypeDerivedFrom	Create query to select stereotype derived from qualified name
HasStereotype	Create query to select architecture elements with stereotype based on specified sub-constraint
HasPort	Create query to select architecture elements with port on component based on specified sub-constraint
HasInterface	Create query to select architecture elements with interface on port based on specified sub-constraint
HasInterfaceElement	Create query to select architecture elements with interface element on interface based on specified sub-constraint
IsInRange	Create query to select range of property values
Property	Create query to select non-evaluated values for object properties or stereotype properties for elements
PropertyValue	Create query to select property from object or stereotype property and then evaluate property value

Examples

Find Elements in a Model Using Queries

This example shows how to find components in a System Composer model using queries.

Open the model.

```
import systemcomposer.query.*;

scKeylessEntrySystem
zcModel = systemcomposer.loadModel('KeylessEntryArchitecture');
```

Find all the software components in the system.

```
con1 = HasStereotype(Property("Name") == "SoftwareComponent");
[compPaths, compObjs] = zcModel.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=1×5 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 = zcModel.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 Lo
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Sensor/Detect Door 
    {'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
```

Find all the base components in the system.

```
con2 = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.BaseComponent"));
baseComps = zcModel.find(con2)
```

```
baseComps = 18x1 cell
    {'KeylessEntryArchitecture/Engine Control System/Start//Stop Button'                }
    {'KeylessEntryArchitecture/Sound System/Dashboard Speaker'                          }
    {'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/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/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 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 = zcModel.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 5ms.

```
con4 = PropertyValue("AutoProfile.SoftwareComponent.WCET") <= 5;
zcModel.find(con4)
```

```
ans = 1x1 cell array
    {'KeylessEntryArchitecture/Sound System/Sound Controller'}
```

```
% You can specify units and it will do the conversions for you
con5 = PropertyValue("AutoProfile.SoftwareComponent.WCET") <= Value(5, 'ms');
query1Comps = zcModel.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 has a cost greater than 10 USD.

```
con6 = PropertyValue("AutoProfile.SoftwareComponent.WCET") > Value(1, 'ms') | PropertyValue("Auto
query2Comps = zcModel.find(con6)
```

```
query2Comps = 2x1 cell
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller'}
```

Close the model.

```
zcModel.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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• “Create Architecture Views Interactively” • “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.	“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 a Model Using Queries”

See Also

`createView` | `find` | `modifyQuery` | `removeQuery` | `runQuery`

Topics

“Create Architectural Views Programmatically”

Introduced in R2019b

systemcomposer.view.BaseViewComponent

(Removed) Base class for 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

This class inherits from the `systemcomposer.view.ViewElement` class.

Properties

`Name` — Name of view component
character vector

Name of view component, returned as a character vector.

Example: `name = get(objBaseViewComponent,'Name')`

Example: `set(objBaseViewComponent,'Name',name)`

`Parent` — Handle to parent view architecture of component
view architecture object

Handle to the parent view architecture of component, returned as a `systemcomposer.view.ViewArchitecture` object.

Example: `parent = get(objBaseViewComponent,'Parent')`

`Architecture` — Handle to view architecture of component
view architecture object

Handle to the view architecture of component, returned 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 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

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

Topics
"Create Architecture Views Interactively"
"Create Architectural Views Programmatically"

Introduced in R2019b

systemcomposer.view.ComponentOccurrence

(Removed) Class that represents 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 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

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

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

Introduced in R2019b

systemcomposer.view.ElementGroup

Class that represents architecture view element group

Description

Use the `ElementGroup` class 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 and its `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 an 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 a 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

Architecture Views in System Composer with Keyless Entry System

This example shows how to use a keyless entry system to programmatically create architecture views using API.

1. Import the package with the queries.

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

2. Open the Simulink® project file for the Keyless Entry System.

```
scKeylessEntrySystem
```

3. Load the example model into System Composer™.

```
zcModel = systemcomposer.loadModel('KeylessEntryArchitecture');
```

Example 1: Hardware Component Review Status View

Create a filtered view that selects all of the hardware components in the architecture model and groups them using the `ReviewStatus` property.

1. Construct the query to select all of the hardware components.

```
hwCompQuery = HasStereotype(IsStereotypeDerivedFrom('AutoProfile.HardwareComponent'))
```

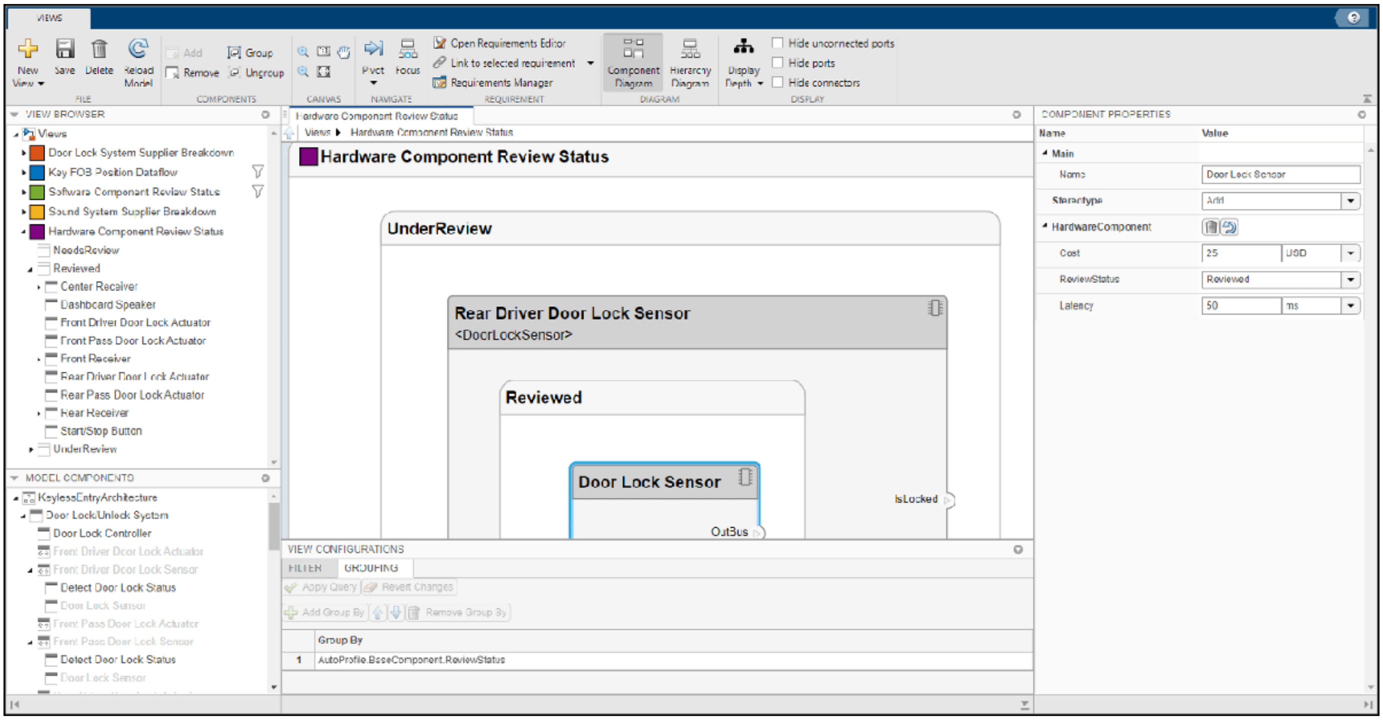
```
hwCompQuery = 
  HasStereotype with properties:

    AllowedParentConstraints: {1x3 cell}
               SubConstraint: [1x1 systemcomposer.query.IsStereotypeDerivedFrom]
              SkipValidation: 0
```

2. Use the query to create a view.

```
zcModel.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. Open the Architecture Views Gallery.

```
zcModel.openViews
```



Example 2: FOB Locator System Supplier View

This example shows how to create a freeform view that manually pulls the components from the FOB Locator System and then 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 = zcModel.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.

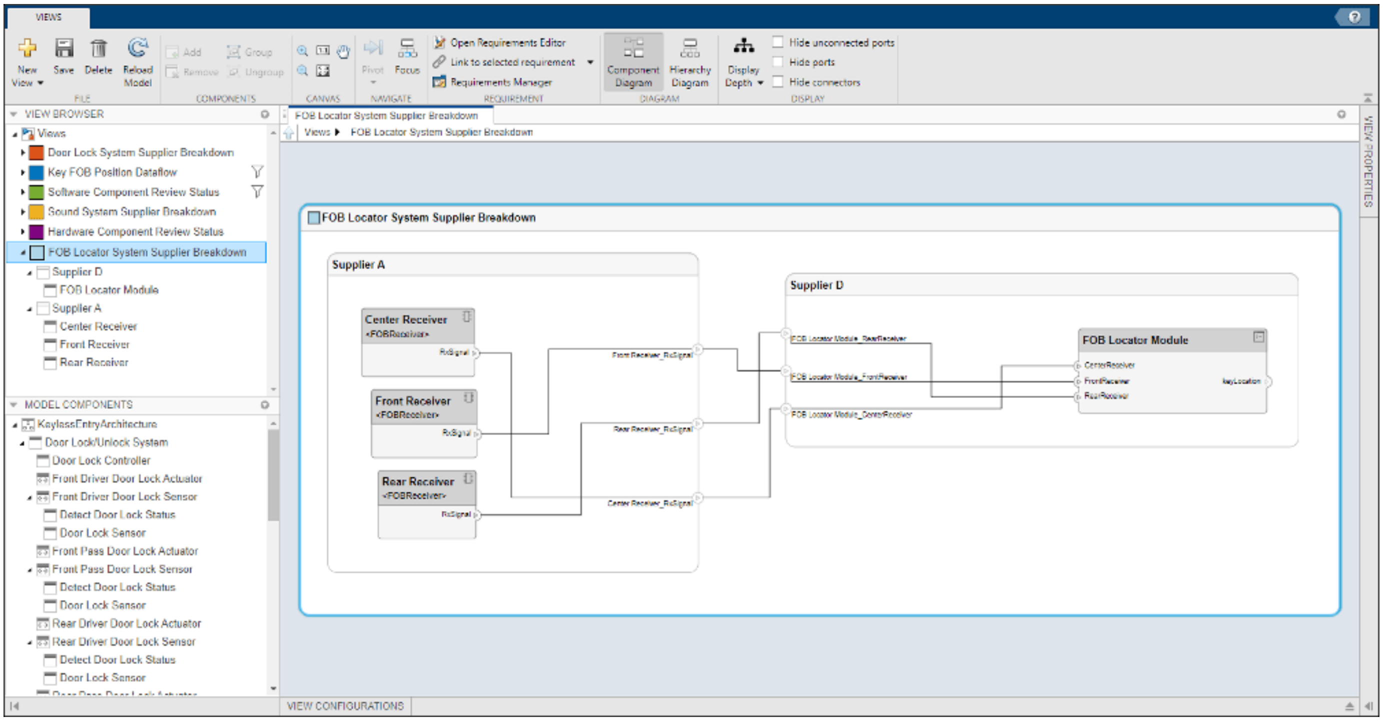
```
supplierD = fobSupplierView.Root.createSubGroup('Supplier D');
supplierD.addElement('KeylessEntryArchitecture/FOB Locator System/FOB Locator Module');
```

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 = zcModel.lookup('Path','KeylessEntryArchitecture/FOB Locator System');

% Find all the components which contain the name "Receiver"
receiverCompPaths = zcModel.find(...
    contains(systemcomposer.query.Property('Name'),'Receiver'),...
    FOBLocatorSystem.Architecture);

supplierA.addElement(receiverCompPaths)
```



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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "Modeling System Architecture of Keyless Entry System"

Term	Definition	Application	More Information
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	"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 a Model Using Queries"

See Also

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

Topics

"Create Architecture Views Interactively"
"Create Architectural Views Programmatically"
"Display Component Hierarchy Using Hierarchy Views"

Introduced in R2021a

systemcomposer.view.View

Class that represents architecture view

Description

Use the `View` class 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 the view, specified as a `systemcomposer.view.ElementGroup` object.

`Model` — Architecture model
model object

Architecture model where the view belongs, specified as a `systemcomposer.arch.Model` object.

`UUID` — Universal unique identifier
character vector

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

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

Data Types: `char`

`Select` — Selection query
constraint object

Selection query associated with a 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 a name `'blue'`, `'black'`, or `'green'` or 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 with values 1 (`true`) or 0 (`false`).

Example: `included = get(objView,'IncludeReferenceModels')`

Data Types: `logical`

Object Functions

modifyQuery	Modify architecture view query and property groupings
runQuery	Re-run architecture view query on model
removeQuery	Remove architecture view query
destroy	Remove model element

Examples

Architecture Views in System Composer with Keyless Entry System

This example shows how to use a keyless entry system to programmatically create architecture views using API.

1. Import the package with the queries.

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

2. Open the Simulink® project file for the Keyless Entry System.

```
scKeylessEntrySystem
```

3. Load the example model into System Composer™.

```
zcModel = systemcomposer.loadModel('KeylessEntryArchitecture');
```

Example 1: Hardware Component Review Status View

Create a filtered view that selects all of the hardware components in the architecture model and groups them using the `ReviewStatus` property.

1. Construct the query to select all of the hardware components.

```
hwCompQuery = HasStereotype(IsStereotypeDerivedFrom('AutoProfile.HardwareComponent'))
```

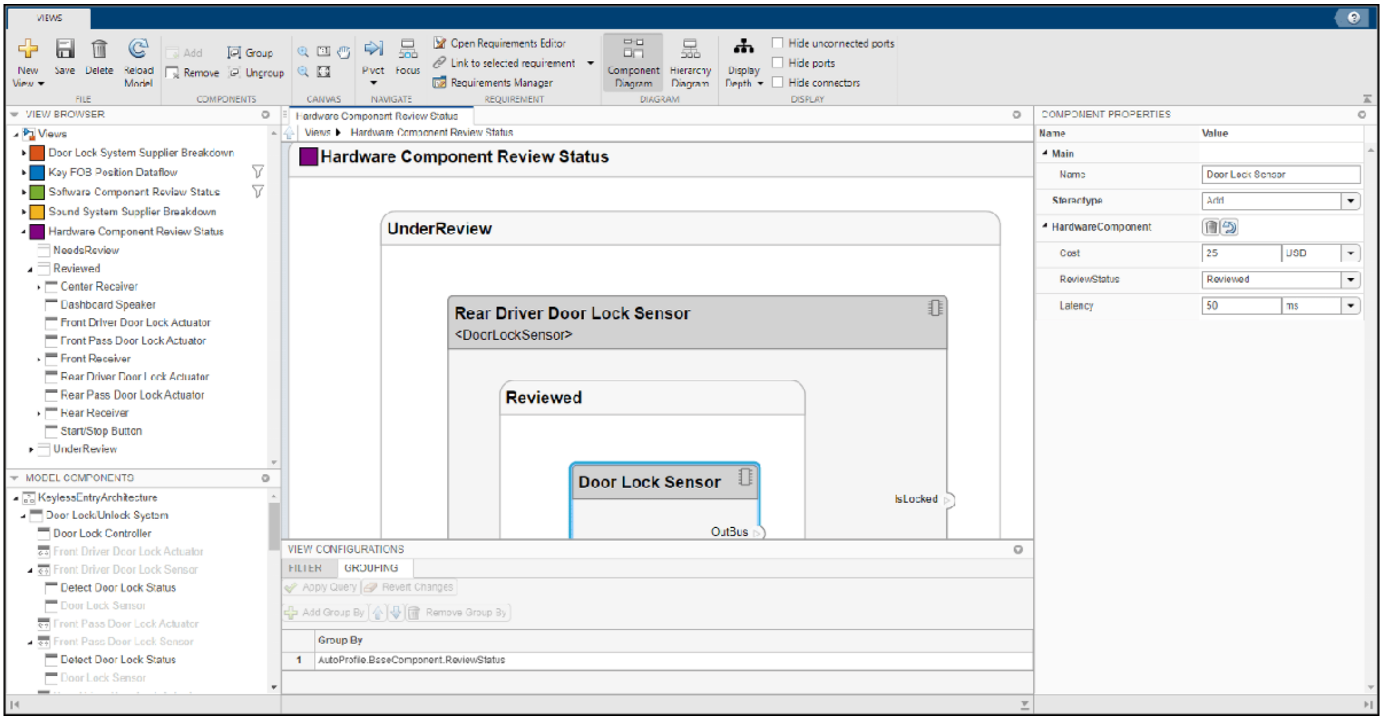
```
hwCompQuery = 
  HasStereotype with properties:

    AllowedParentConstraints: {1x3 cell}
               SubConstraint: [1x1 systemcomposer.query.IsStereotypeDerivedFrom]
              SkipValidation: 0
```

2. Use the query to create a view.

```
zcModel.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. Open the Architecture Views Gallery.

```
zcModel.openViews
```



Example 2: FOB Locator System Supplier View

This example shows how to create a freeform view that manually pulls the components from the FOB Locator System and then 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 = zcModel.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.

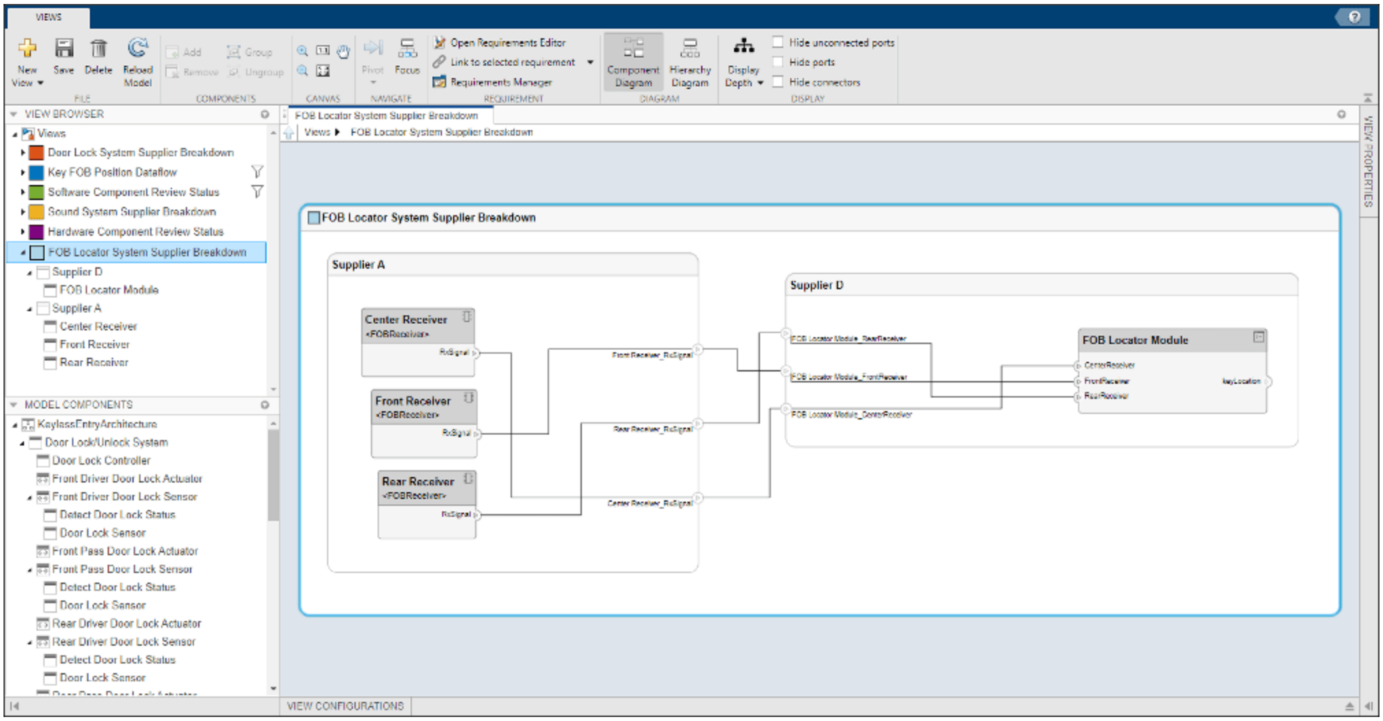
```
supplierD = fobSupplierView.Root.createSubGroup('Supplier D');
supplierD.addElement('KeylessEntryArchitecture/FOB Locator System/FOB Locator Module');
```

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 = zcModel.lookup('Path','KeylessEntryArchitecture/FOB Locator System');

% Find all the components which contain the name "Receiver"
receiverCompPaths = zcModel.find(...
    contains(systemcomposer.query.Property('Name'),'Receiver'),...
    FOBLocatorSystem.Architecture);

supplierA.addElement(receiverCompPaths)
```



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. Construct views by pulling in 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: • *Operational views* demonstrate how a system will be used and should be well integrated with requirements analysis. • *Functional views* focus on what the system must do to operate. • *Physical views* show how the system is constructed and configured. A viewpoint represents a stakeholder perspective that specifies the contents of the view.	• "Create Architecture Views Interactively" • "Modeling System Architecture of Keyless Entry System"

Term	Definition	Application	More Information
element group	An element group is a grouping of components in a view.	Use element groups to programmatically populate a view.	"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 a Model Using Queries"

See Also

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

Topics

"Create Architecture Views Interactively"
"Create Architectural Views Programmatically"
"Display Component Hierarchy Using Hierarchy Views"

Introduced in R2021a

systemcomposer.view.ViewArchitecture

(Removed) Class that represents 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` 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 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

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

Topics

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

Introduced in R2019b

systemcomposer.view.ViewComponent

(Removed) Class that represents 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` is a component that exists only in the view it is created in. 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 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

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

Topics
"Create Architecture Views Interactively"
"Create Architectural Views Programmatically"

Introduced in R2019b

systemcomposer.view.ViewElement

(Removed) Base class of 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

Gets the identifier of an object. 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 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

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

Topics
"Create Architecture Views Interactively"
"Create Architectural Views Programmatically"

Introduced in R2009b

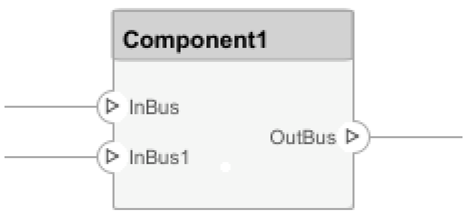
Blocks

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 for connecting 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 from the **Modeling** tab or the palette. You can also click and drag a box on the canvas, selecting the Component option once complete.
- Add a port by selecting an edge of the component and choosing a direction from the menu: `Input` or `Output`
- Click and drag the port to create a connection. Connect to another component, or have the option of creating a new component to complete the connection.
- To connect the architecture 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.

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"
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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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

Functions
`addComponent` | `addPort` | `connect`

Blocks
Adapter | Reference Component | Variant Component

Topics
"Create an Architecture Model"

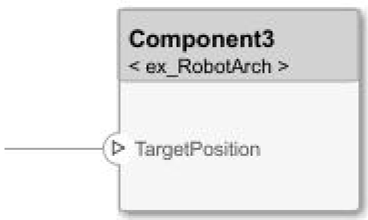
Introduced in R2019a

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 from the **Modeling** tab or the palette. You can also click and drag a box on the canvas, selecting the Reference Component option once complete.
- Add a port by selecting an edge of the component and choosing a direction from the menu: `Input` or `Output`
- Click and drag the port to create a connection. Connect to another component, or have the option of creating a new component to complete the connection.
- To connect the architecture 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 contents:

- Upon creating a Reference Component, you have the option to right-click on 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 into 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, you have the option to right-click and select `Inline Model` which inlines the contents of the architecture model referenced by the specified component and breaks the link to the reference model. The Reference Component 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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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”

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	"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.	• "Implement Component Behavior in Simulink" • "Create a Reference Architecture"

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 an architectural component using state machines.	"Add Stateflow Chart Behavior to Architecture Component"
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.	• "Define Sequence Diagrams" • "Use Sequence Diagrams in the Views Gallery"

See Also

Functions
`addComponent` | `addPort` | `connect` | `createSimulinkBehavior` | `inlineComponent` | `isReference` | `linkToModel` | `saveAsModel`

Blocks
Adapter | Component | Variant Component

Topics
"Implement Component Behavior in Simulink"

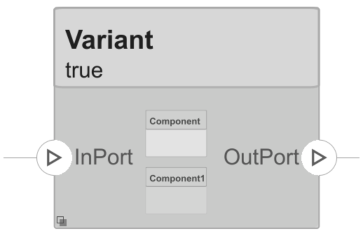
Introduced in R2019a

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 from the **Modeling** tab or the palette. You can also click and drag a box on the canvas, selecting the Variant Component option once complete.
- Add a port by selecting an edge of the component and choosing a direction from the menu: `Input` or `Output`
- Click and drag the port to create a connection. Connect to another component, or have the option of creating a new component to complete the connection.
- To connect the architecture 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 upon Variant Component creation. Right-click on the variant component and select **Variant > Label Mode Active Choice** and choose the active choice.
- To add an additional variant choice, right-click on the variant component and select **Variant > Add Variant Choice**.
- Double-click into the Variant Component to design the variants within it.

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: • *Functional architecture* describes the flow of data in a system. • *Logical architecture* describes the intended operation of a system. • *Physical architecture* 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: • 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"

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: • *Component ports* are interaction points on the component to other components. • *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.	“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 1-417

See Also

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

Blocks
Adapter | Component | Reference Component

Topics
"Decompose and Reuse Components"

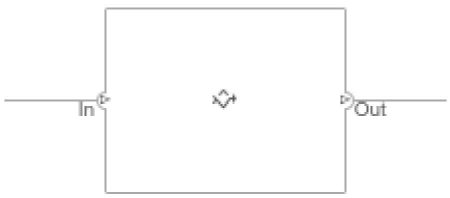
Introduced in R2019a

Adapter

Connect components with different interfaces

Description

The Adapter block allows you to adapt dissimilar interfaces. 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 block comes with an `In` and `Out` port.
- Click and drag a port to create a connection. Connect each port to another component, or have the option of creating a new component to complete the connection.

To use an Adapter block:

- Insert an adapter block between two ports with different interfaces which need to communicate. You will be able to create mappings between interface elements on each port.
- Double-click on the Adapter block to open up 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	An interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. An interface can be composite, meaning that it can include elements that describe the properties of an interface signal.	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 interfaces and interface elements and store them in an interface data dictionary for reuse between models.	"Define Interfaces"
interface element	An interface element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.	Interface elements describe the decompositions of an interface: • Pins or wires in a connector or harness. • Messages transmitted across a bus. • Data structures shared between components.	"Assign Interfaces to Ports"
interface dictionary	An interface data dictionary is a consolidated list of all the interfaces 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 and interface elements. Data dictionaries are stored in separate `.sldd` files.	• "Save, Link, and Delete Interfaces" • "Reference Data Dictionaries"

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.	With an adapter, you can perform three functions on the Interface Adapter dialog: • Create and edit mappings between input and output interfaces. • Apply an interface conversion `UnitDelay` to break an algebraic loop. • Apply an interface conversion `RateTransition` to reconcile different sample time rates for reference models.	"Interface Adapter"

See Also

Functions
`connect`

Blocks
Component | Reference Component | Variant Component

Topics
"Assign Interfaces to Ports"
"Interface Adapter"

Introduced in R2019a

Sequence Viewer

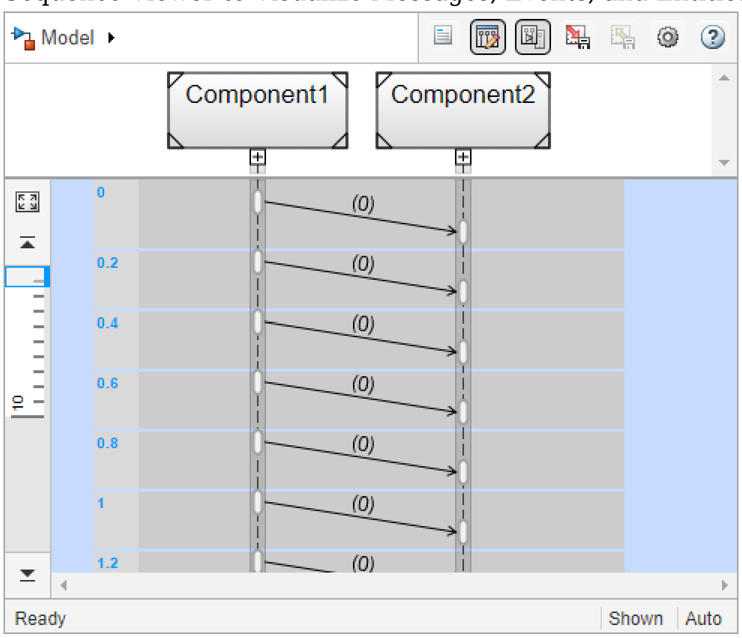
Visualize messages, events, states, transitions, and functions

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".



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