System Composer™

User's Guide

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™ User's Guide

Trademarks

Patents

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)

1 Architecture Model Editing

2 Requirements

3 Interface Management

4 Define Architectural Properties

5 Use Simulink Models with System Composer

6 Analyze Architecture Model

7 Software Architectures

Architecture Model Editing

Compose Architecture Visually

In this section...

Create and edit visual diagrams to represent system architecture in System Composer™. Use visual architecture elements, components, ports, and connections in the system composition. Model hierarchy in architecture by decomposing components. Navigate through the hierarchy.

Create an Architecture Model

Start with a blank architecture model to model physical and logical architecture of a system.

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.

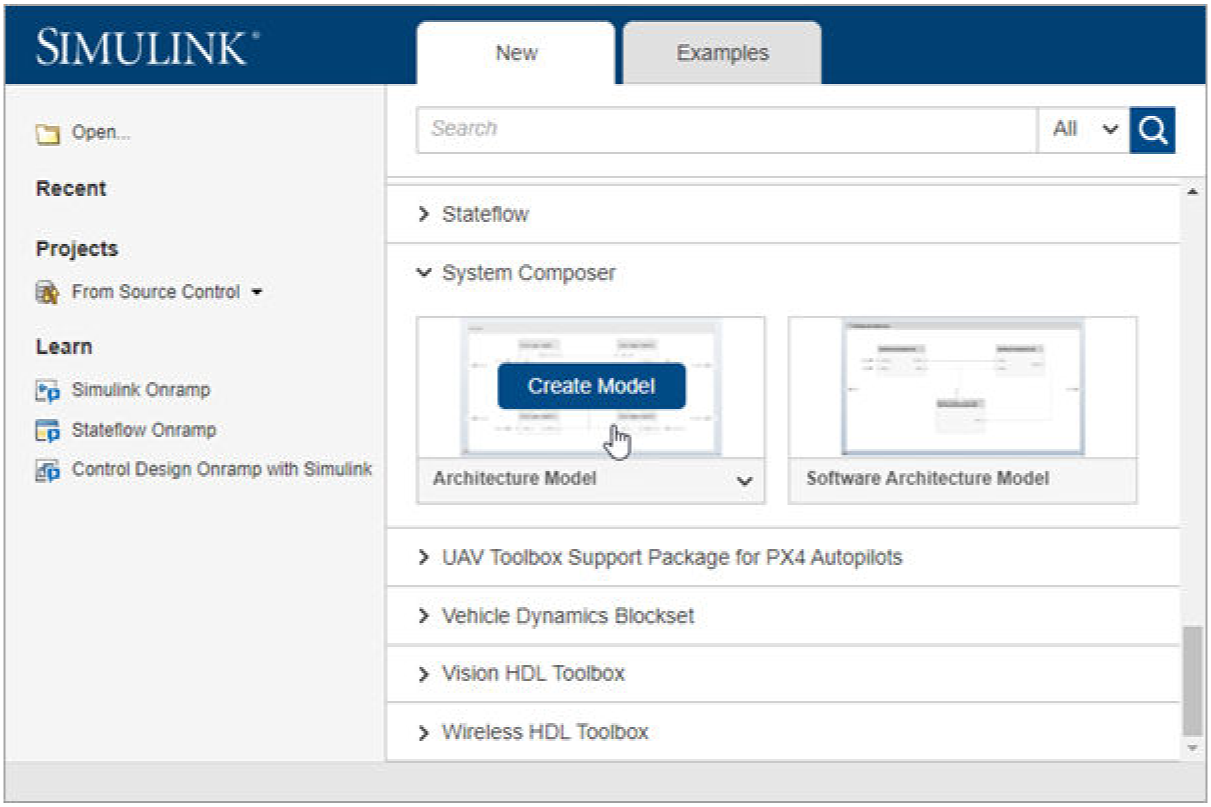
A System Composer model is the `.slx` file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.

An architecture model includes a top-level architecture that holds the composition of the system. This top-level architecture also allows definition of interfaces of this system with other systems. Use one of these methods to create an architecture model:

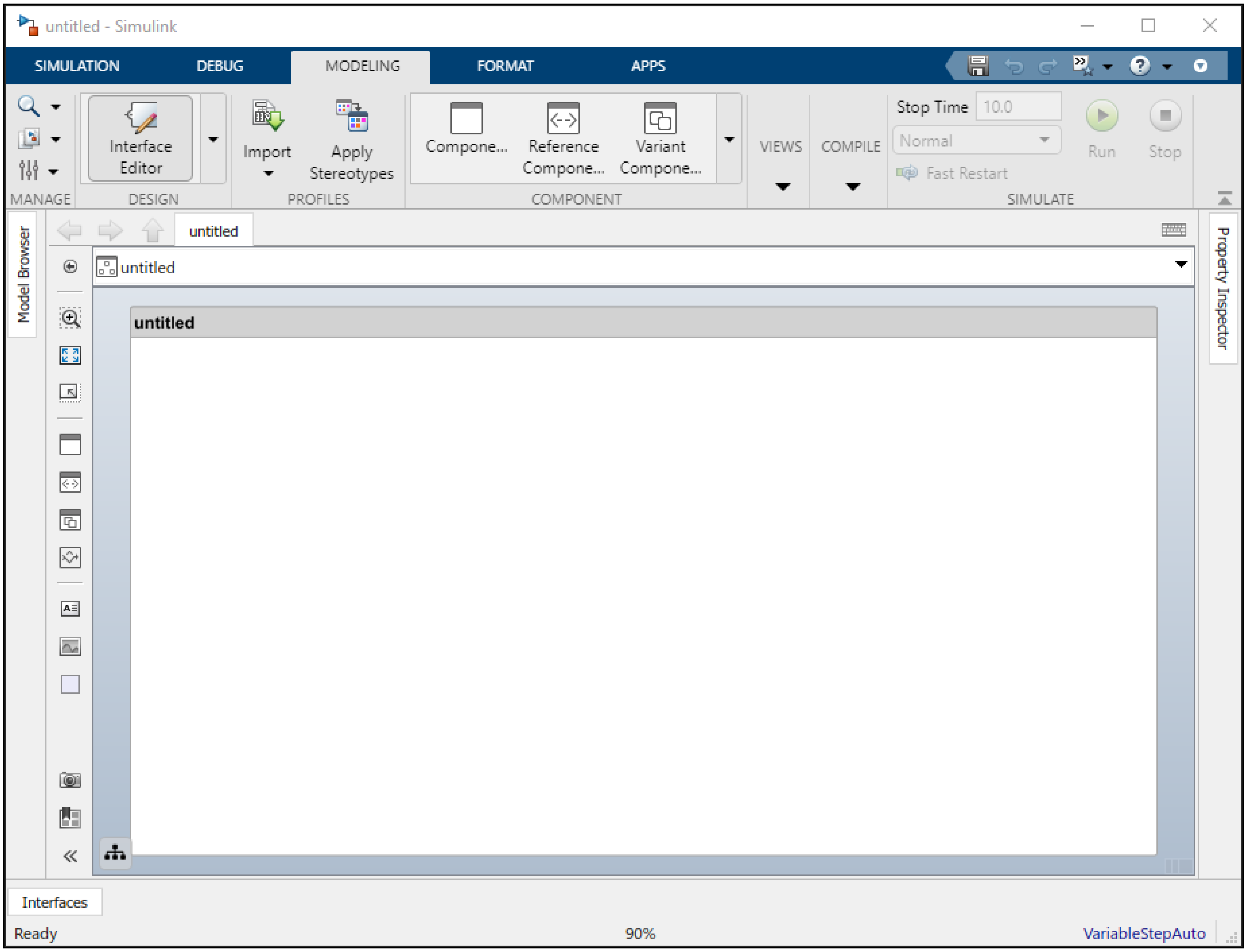
- At the command line, type

  ```
  systemcomposer
  ```

  Select **Architecture Model**.



- From a Simulink® model or a System Composer architecture model. On the Simulation tab, select New, and then select Architecture.



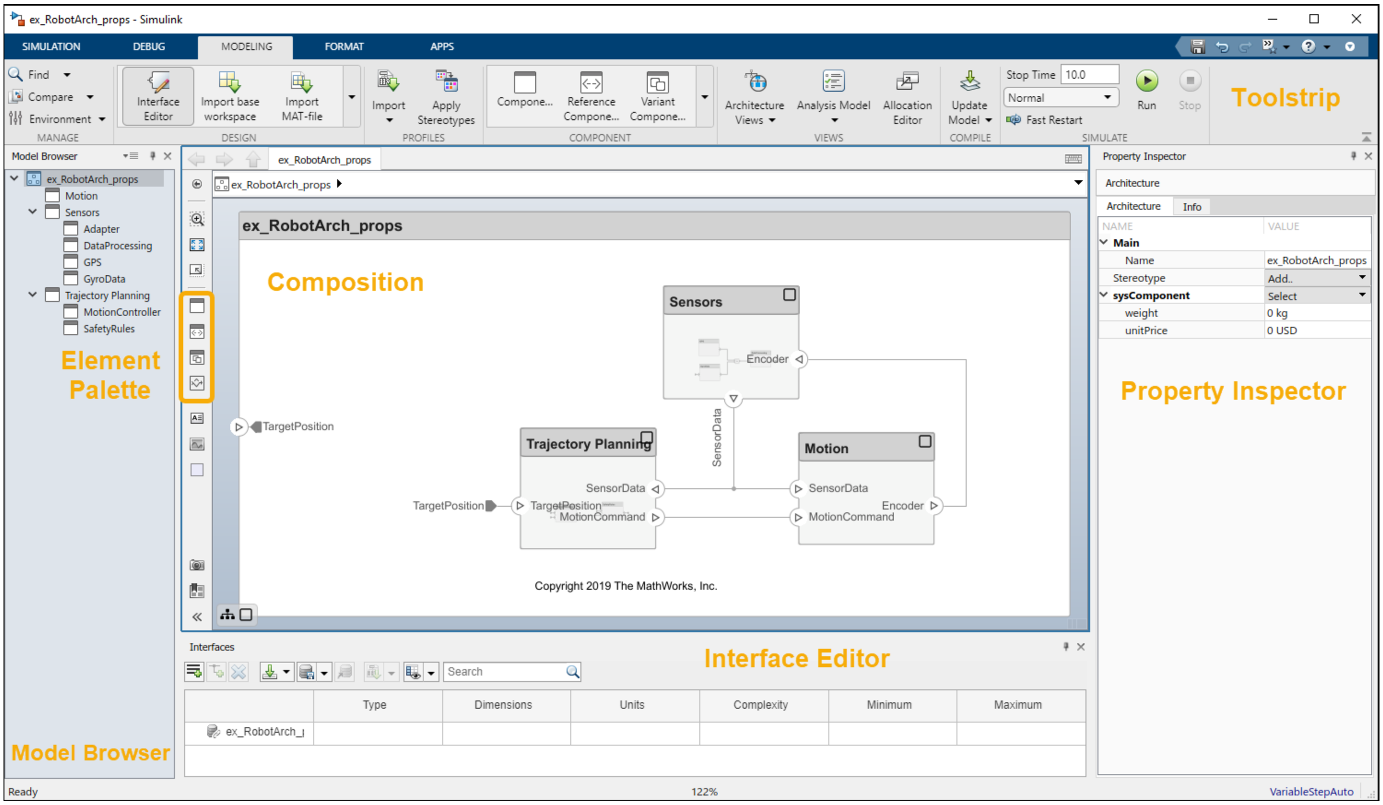
- At the MATLAB® command line, type:

  ```
  archModel = new_system('ModelName','Architecture');
  open_system(archModel)
  ```

  where `ModelName` is the name of the new model.

Save the architecture model. On the **Simulation** tab, select **Save All** 🖫. The architecture model is saved as an `.slx` file.

The architecture model includes a top-level architecture that holds the composition of the system. This top-level architecture also allows definition of interfaces of this system with other systems. The composition represents a structured parts list — a hierarchy of components with their interfaces and interconnections. Edit the composition in the Composition Editor.



This example shows a motion control architecture, where a sensor obtains information from a motor, feeds that information to a controller, which in turn processes this information to send a control signal to the motor so that it moves in a certain way. You can start with this rough description and add component properties, interface definitions, and requirements as the design progresses.

Components

A component is a 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.

The Component element in System Composer can represent a component at any level of the system hierarchy, whether it is a major system component that encompasses many subsystems, such as a controller with its hardware and software, or a component at the lowest level of hierarchy, such as a software module for messaging.

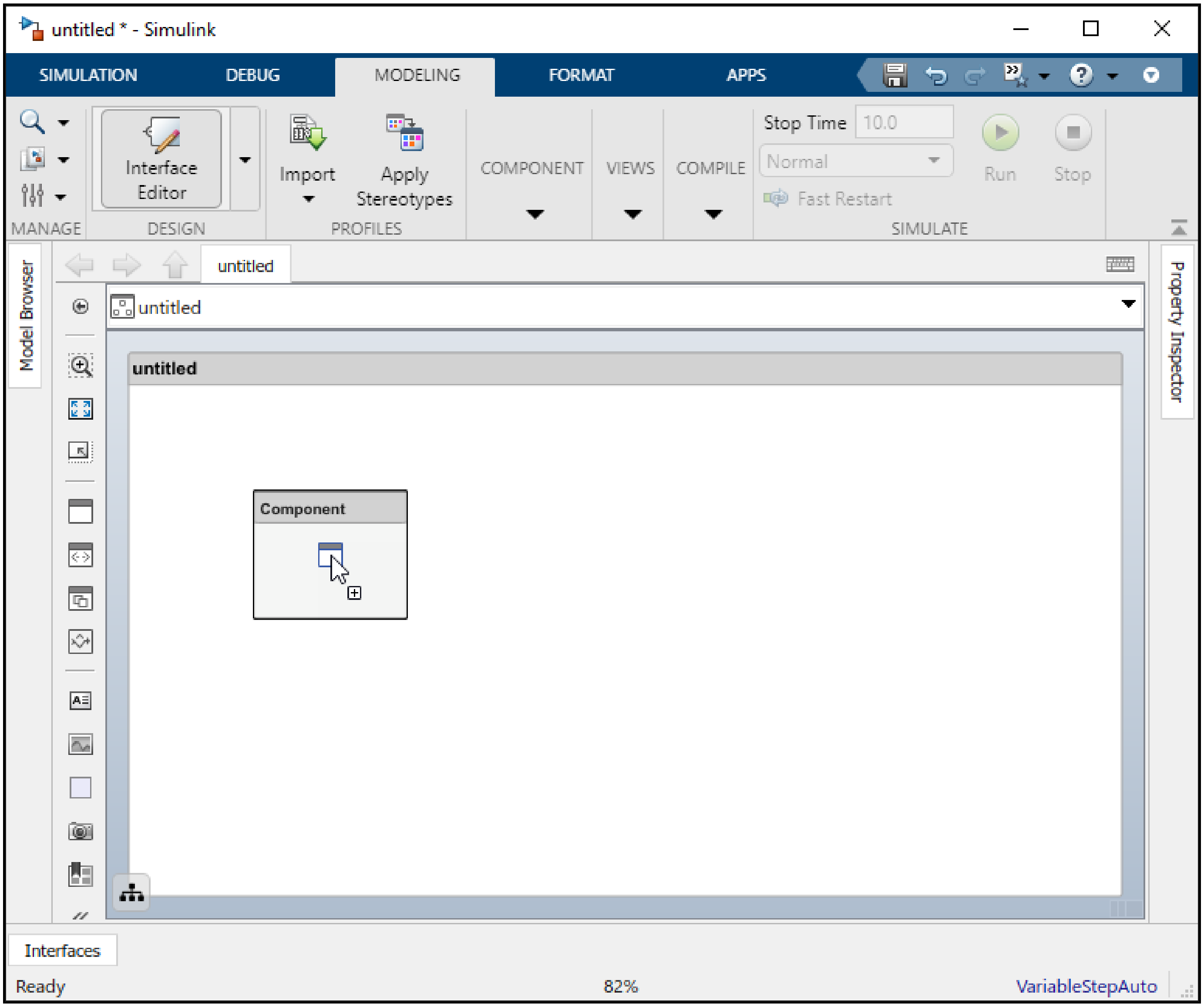
Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts.

Add Components

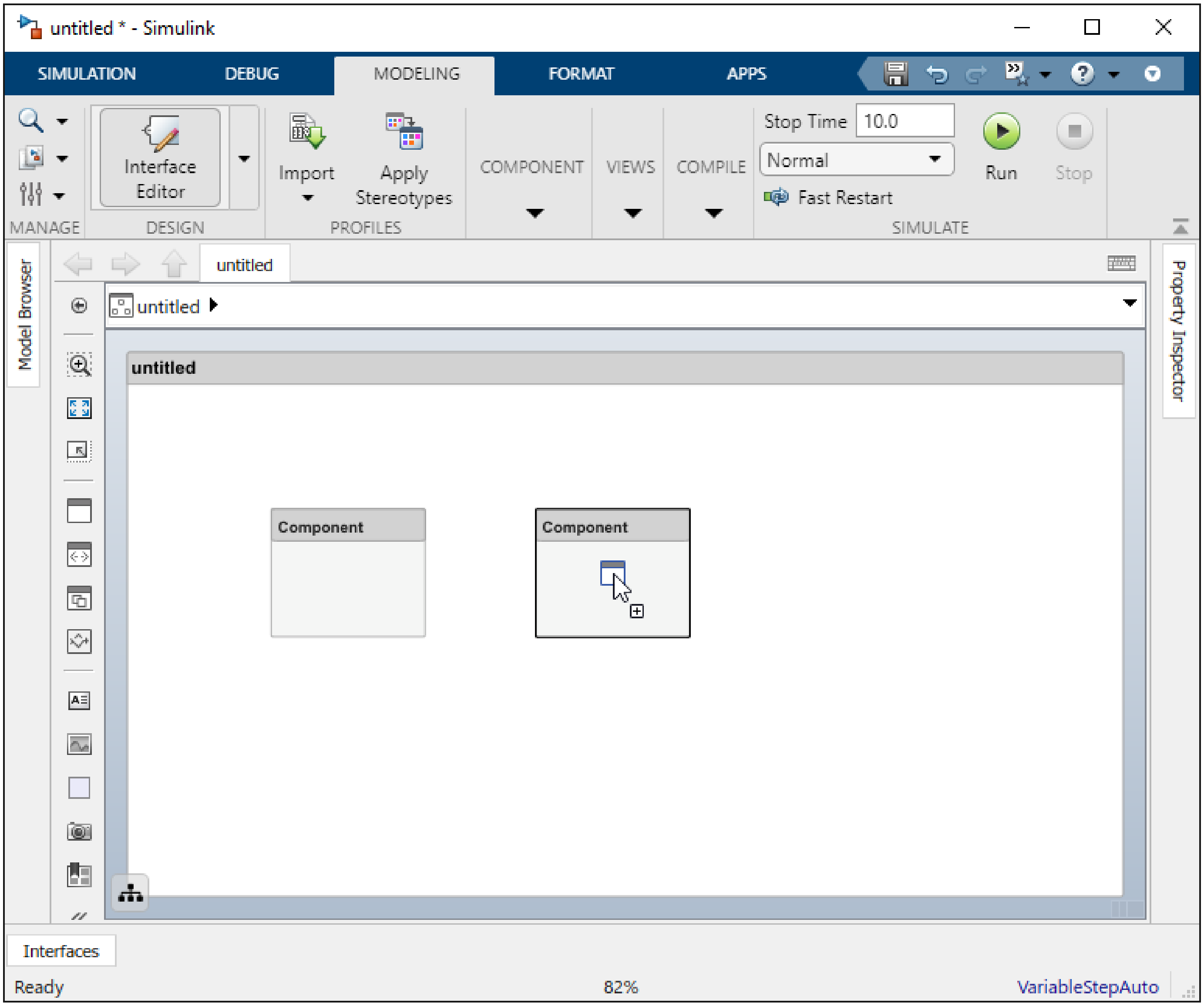
Use one of these methods to add components to the architecture:

- Draw a component — In the canvas, left-click and drag the mouse to create a rectangle. Release the mouse button to see the component outline. Select the Component block option to commit.

- Create a single component from the palette —



- Create multiple components from the palette —

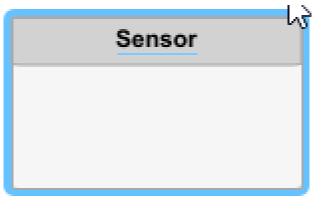


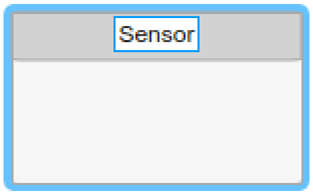
Stop Time 10.0

Normal

Name a Component

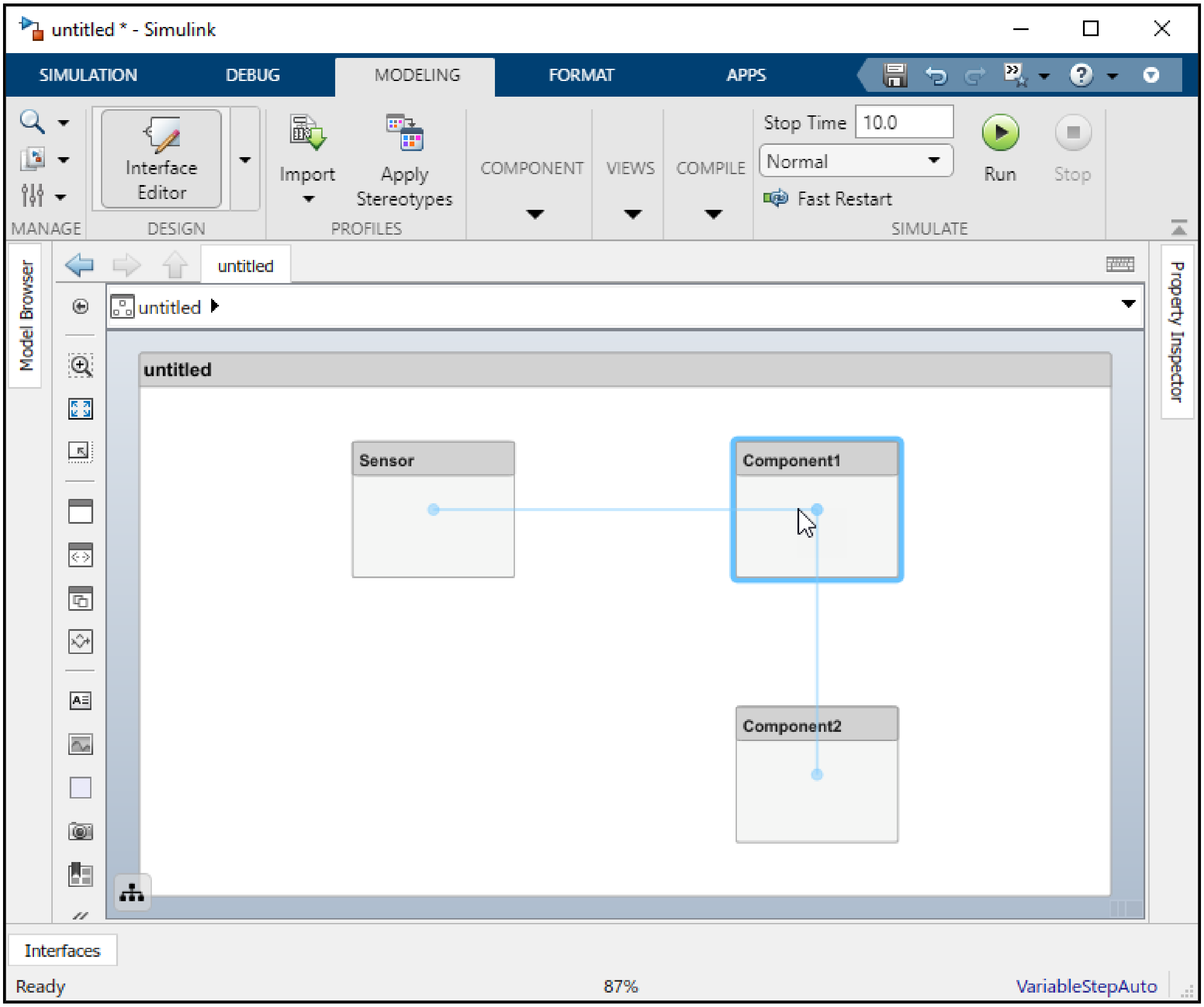
Each component must have a name that is unique within the same architecture level. The name of the component is highlighted upon creation so you can directly type the name. To change the name of a component, click the component and then click its name.





Move a Component

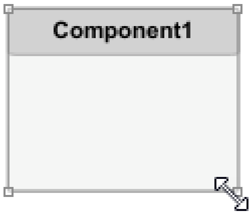
Move a component simply by clicking and dragging it. Blue guidelines may appear to help align the component with other components.



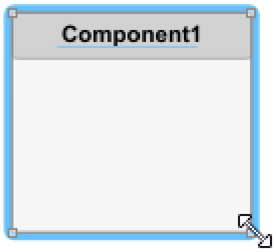
Resize a Component

Resize a component by dragging corners.

1 Hover the pointer over a corner to see the double arrow.



2 Left-click the corner and drag while holding the mouse button down. If you want to resize the component proportionally, hold the **Shift** button as well.



3 Release the mouse button when the component reaches the size you want.

Delete a Component

Click a component and press **Delete** to delete it. To delete multiple components, select them while holding the **Shift** key down, then press **Delete** or right-click and select **Delete** from the context menu.

Ports

A port is a node on a component or 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:

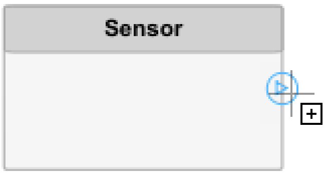
- *Component ports* are interaction points on the component to other components.
- *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.

For example, a sensor might have data ports to communicate with a motor and a controller. Its input port takes data from the motor, and the output port delivers data to the controller. You can specify data properties by defining an interface as described in “Define Interfaces” on page 3-2.

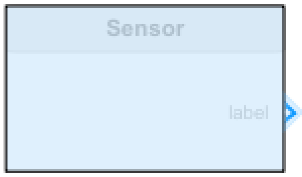
Add a Component Port

Represent the relationship between components by defining directional interface ports. You can organize the diagram by positioning ports on any edge of the component, in any position.

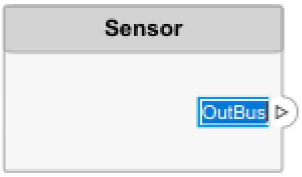
1 Pause over the side of a component. A + sign and a port outline appear.



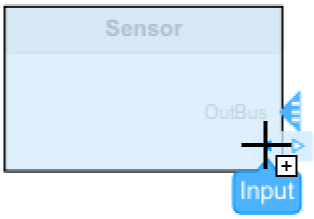
2 Click the port outline. The component is shaded blue and a port arrow appears.



3 Click the arrow to commit the port. You can also name the port at this point.



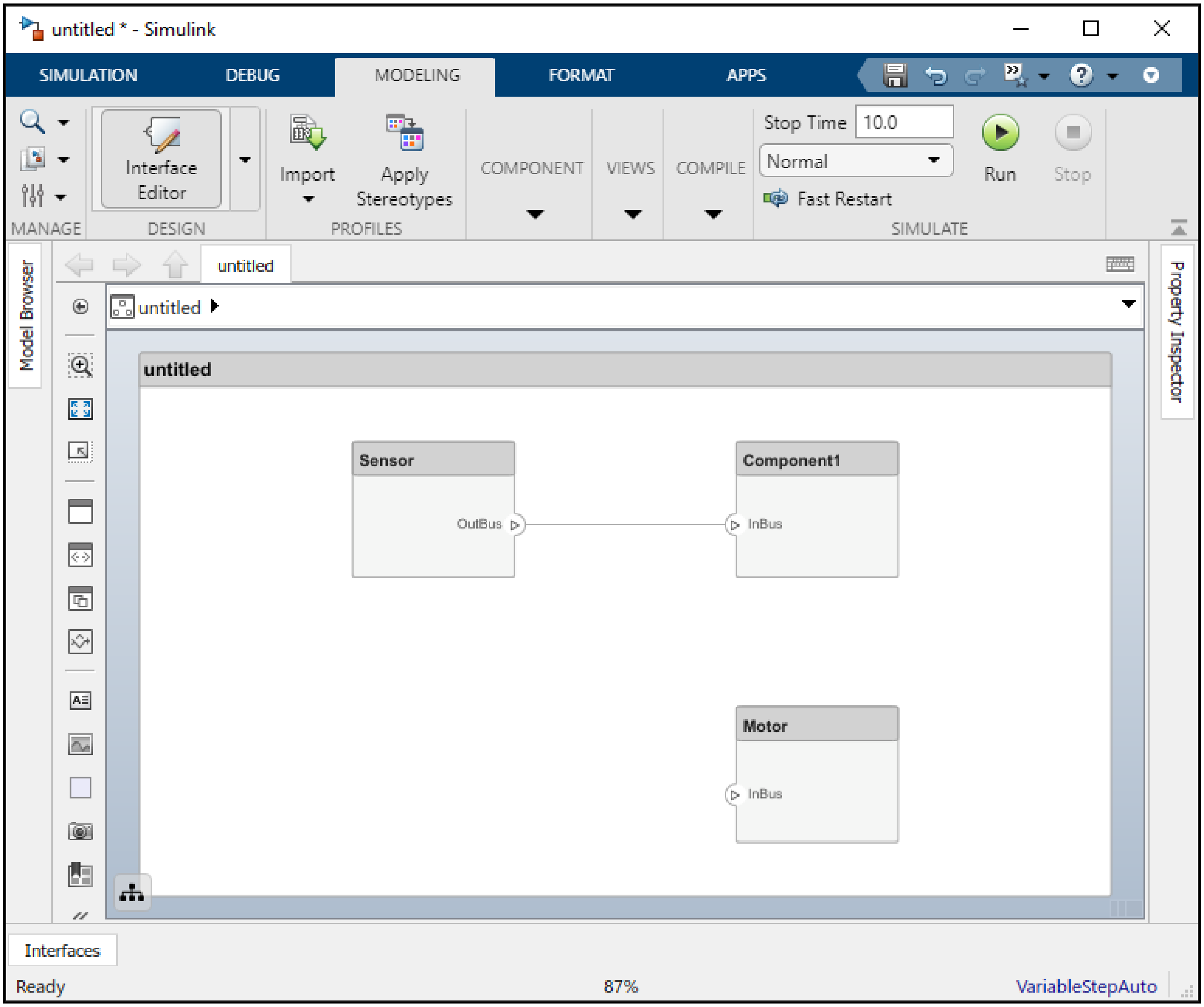
An output port is shown with the ▷ icon and an input port is shown with the ▷ icon. By default, a port created on the top or left edge of a component is an input port, and a port created on the bottom or right edge is an output port. To designate port direction at creation, after you click the edge, pause over the arrow outline to see direction options. Select **Input** or **Output** before committing the port.



You can move any port to any component edge after creation.

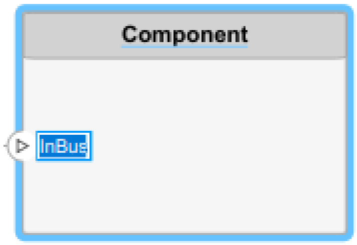
Add an Architecture Port

You can also create a port for the architecture that contains components. These system ports carry the interface of the system with other systems. Pause on any edge of the system box and click when the + sign appears. Click the left side to create input ports and click the right side to create output ports.



Name a Port

Every port is created with a name. To change the name, click it and edit.



Ports of a component must have unique names.

Move a Port

You can move a port to any side of a component. Select the port and use arrow keys.

Arrow Key	Original Port Edge	Port Movement
Up	Left or right	If below other ports on the same edge, move up, if not, move to the top edge
	Top or bottom	No action
Right	Top or bottom	If to the left of other ports on the same edge, move right, if not, move to the right edge
	Left or right	No action
Down	Left or right	If above other ports on the same edge, move down, if not, move to the bottom edge
	Top or bottom	No action
Left	Top or bottom	If to the right of other ports on the same edge, move left, if not, move to the left edge
	Left or right	No action

The spacing of the ports on one side is automatic. There can be a combination of input and output ports on the same edge.

Delete a Port

Delete a port by selecting it and pressing the **Delete** button.

Connections

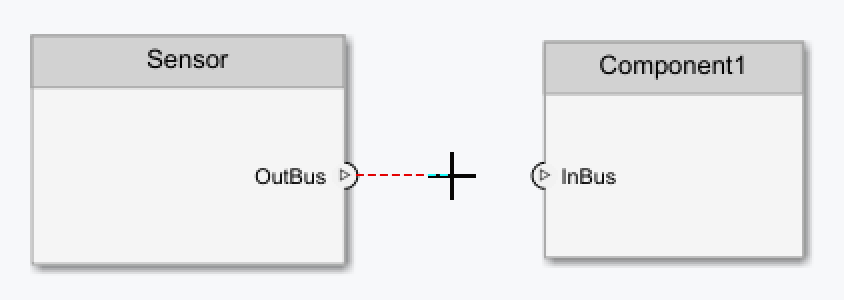
Connectors are lines that provide connections between ports. Connectors describe how information flows between components or architectures. A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.

Connections are visual representations of data flow from an output port to an input port. For example, a connection from a motor to a sensor carries positional information.

Connect Existing Ports

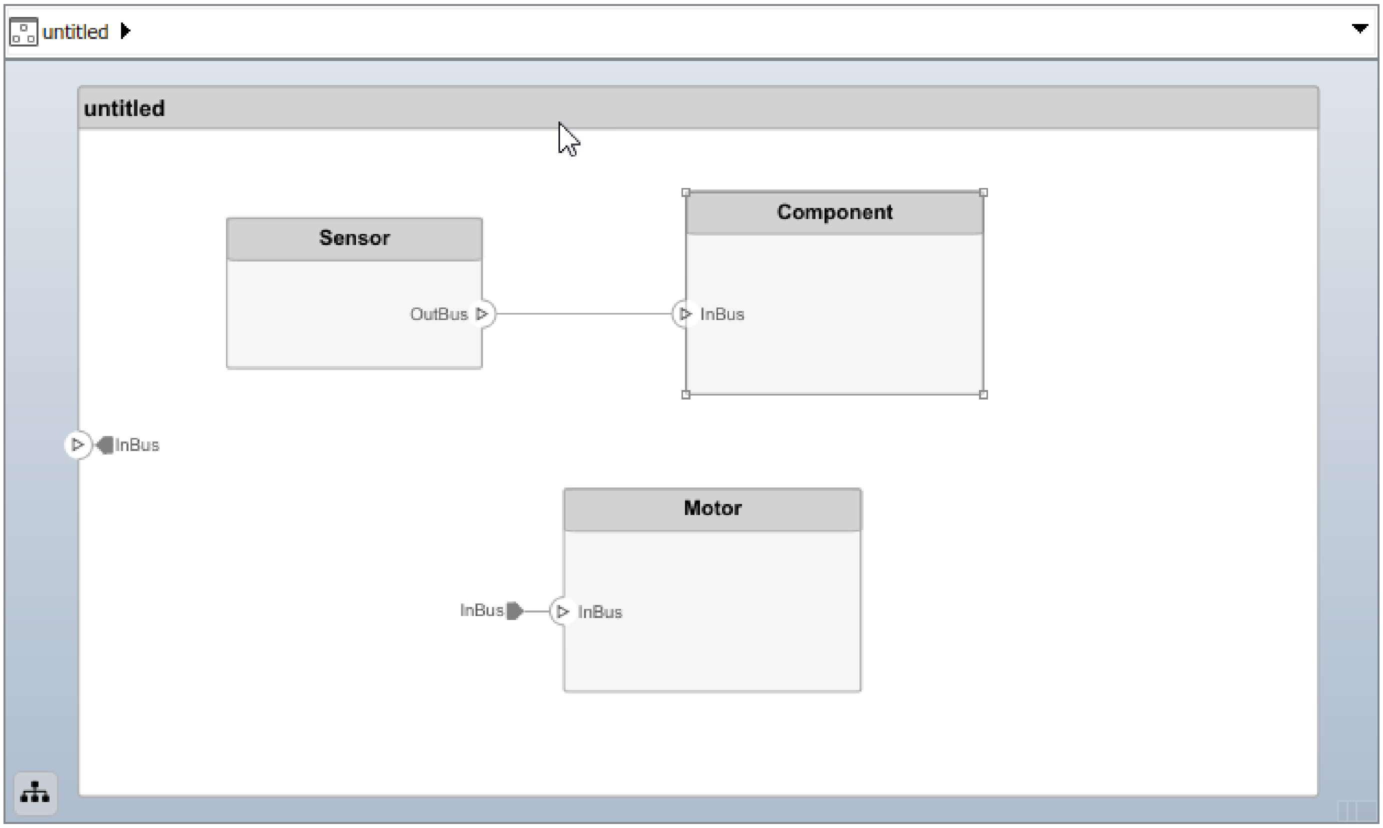
Connect two ports by dragging a line:

1. Click one of the ports.
2. Keep the mouse button down while dragging a line to the other port.
3. Release the mouse button at the destination port. A black line indicates the connection is complete. A red-dotted line appears if the connection is incomplete.



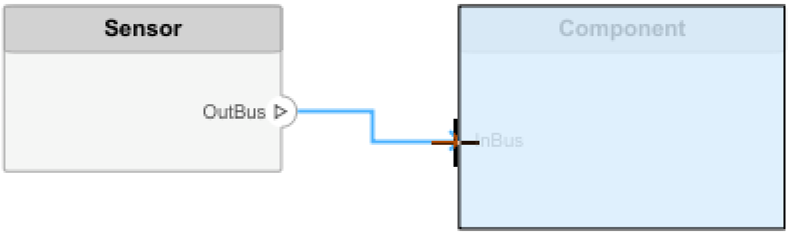
You can take these steps in both directions — input port to output port, or output port to input port. You cannot connect ports that have the same direction.

A connection between an architecture port and a component port is shown with tags instead of lines.



Connect Components Without Ports

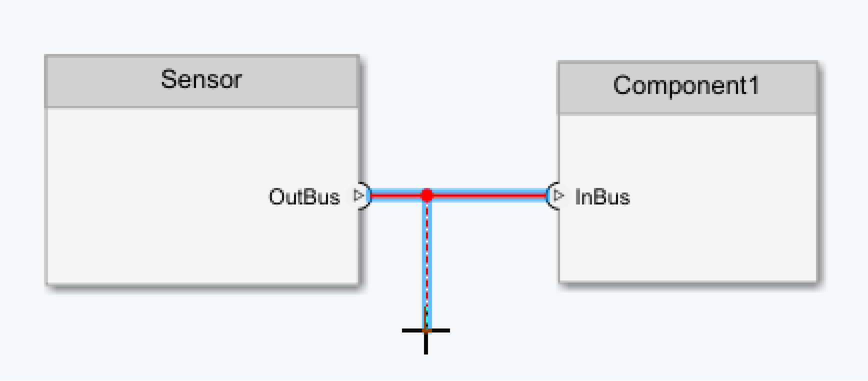
To quickly create ports and connections at the same time, drag a line from one component edge to another. The direction of this connection depends on which edges of the components are used - left and top edges are considered inputs, right and bottom edges are considered outputs. You can also perform this operation from an existing port to a component edge.



You can create a connection between an edge that is assumed to be an input only with an edge that is assumed to be an output. For example, you cannot connect a top edge, which is assumed to be an input, with another top edge, unless one of them already has an output port.

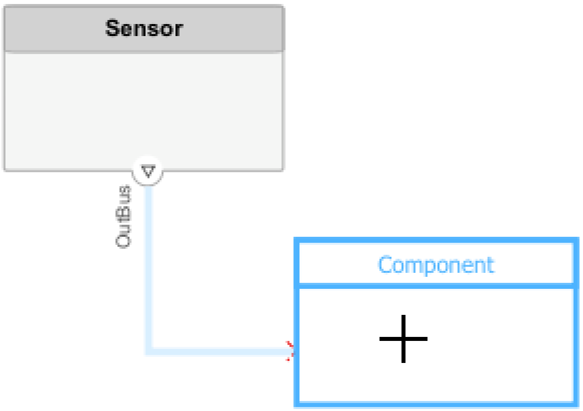
Branch Connections

Connect an output port to multiple input ports by branching a connection. To branch, right-click an existing connection and drag to an input port while holding the mouse button down. Release the button to commit the new connection.



Create New Components Through Connections

If you start a connection from an output port and release the mouse button without a destination port, a new component tentatively appears. Accept the new component by clicking it.



Importing Architectures

By combining the programmatic APIs of System Composer with MATLAB support for loading and parsing many different file and databased formats, you can import external architecture descriptions into System Composer. For details, see “Import and Export Architecture Models” on page 1-50.

You can setup a profile with stereotypes ahead of time to capture the architecture properties represented in such descriptions. For details, see “Define Profiles and Stereotypes” on page 4-2.

Subsequently, you can use MATLAB programming to create and customize the various architectural elements through the set of programmatic APIs. For details, see “Build an Architecture Model from Command Line” on page 1-29.

See Also

Functions
`addComponent` | `addPort` | `connect` | `createModel` | `exportModel` | `importModel`

Blocks
Component

More About

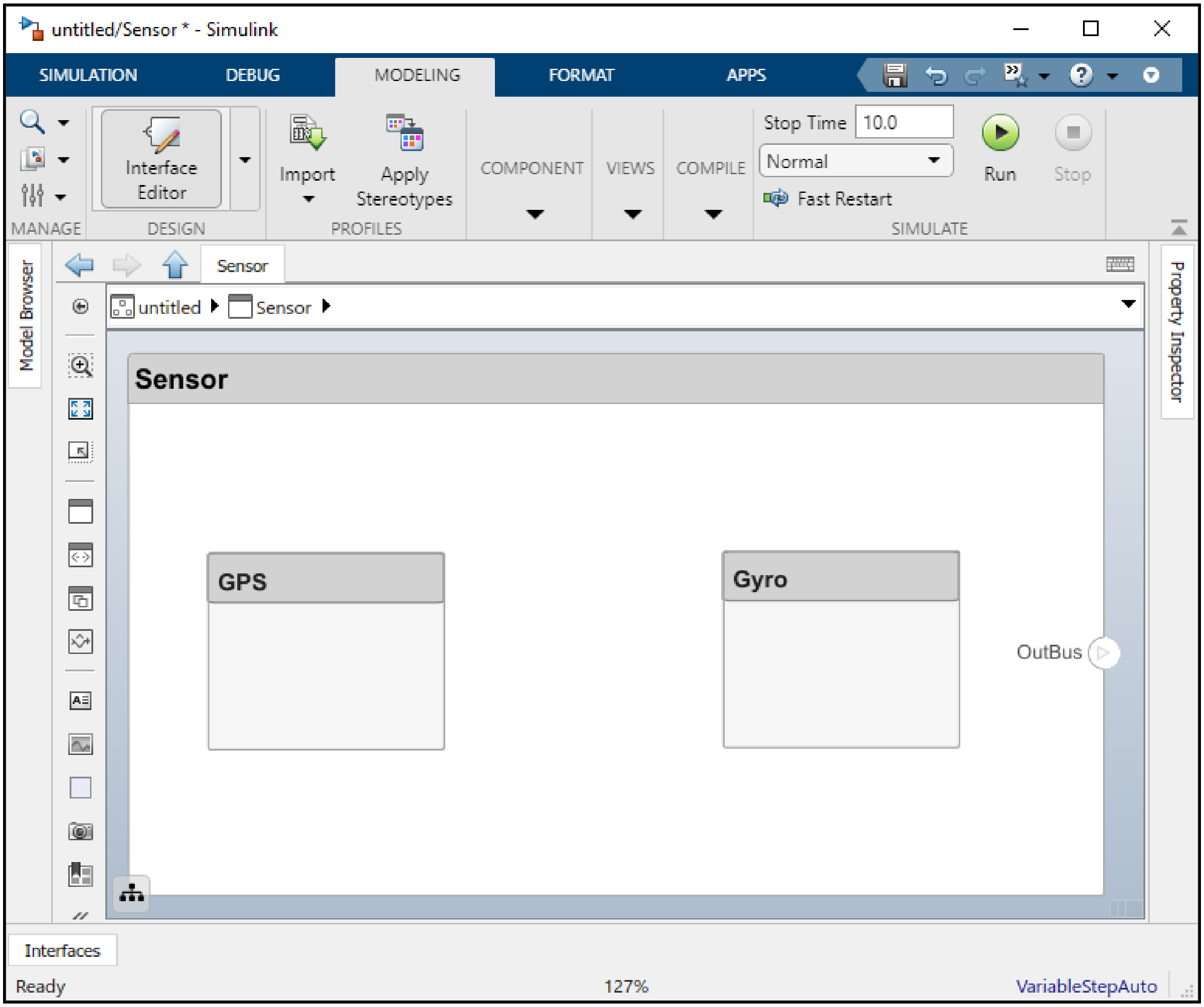
- “Decompose and Reuse Components” on page 1-16
- “Define Interfaces” on page 3-2
- “Assign Interfaces to Ports” on page 3-7

Decompose and Reuse Components

Every component in an architecture model can have its own design, or even several design alternatives. These designs can be architectures modeled in System Composer or behaviors modeled in Simulink. Engineering systems often use the same component design in multiple places. A common component, such as power switch, can be part of all electrical components. You can reuse a component in System Composer within the same model as well as across architecture models.

Decompose a Component

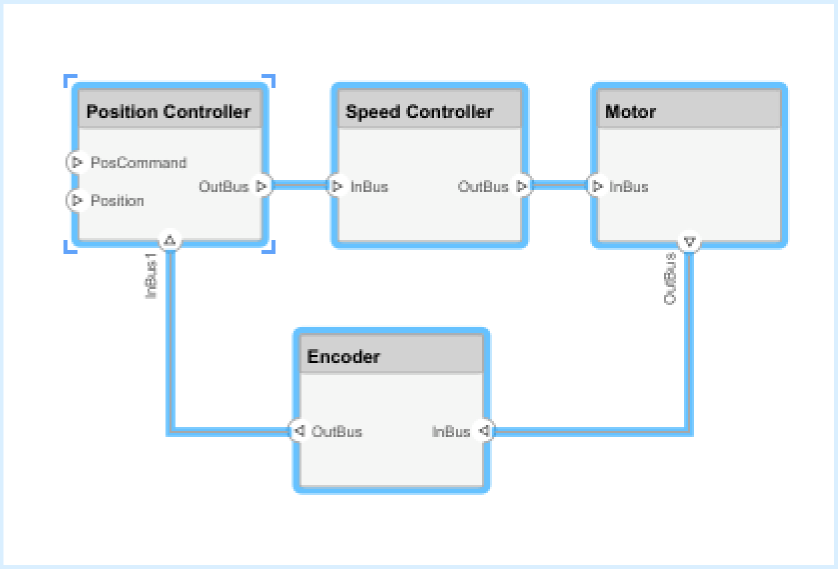
A component can have its own architecture. Double-click a component to view or edit its architecture. When you view the component at this level, its ports appear as architecture ports. You can use the navigation arrows ⇦ ⇨ ⇧ on the toolbar to move through the hierarchy. Use the Model Browser to view component hierarchy.



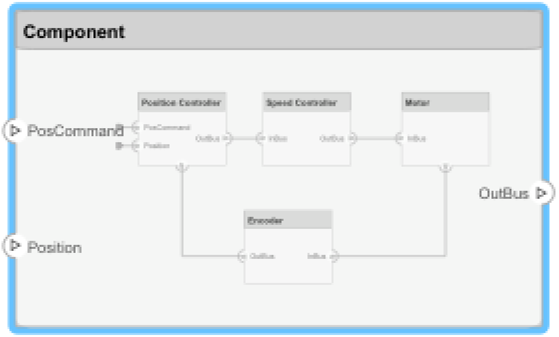
You can add components, ports, and connections at this level to define the architecture.

You can also make a new component from a group of components.

1 Select the components. Either click and drag a rectangle, or select multiple components by holding the **Shift** button down.



2 Create a component from the selected elements using **Architecture > Create Component**



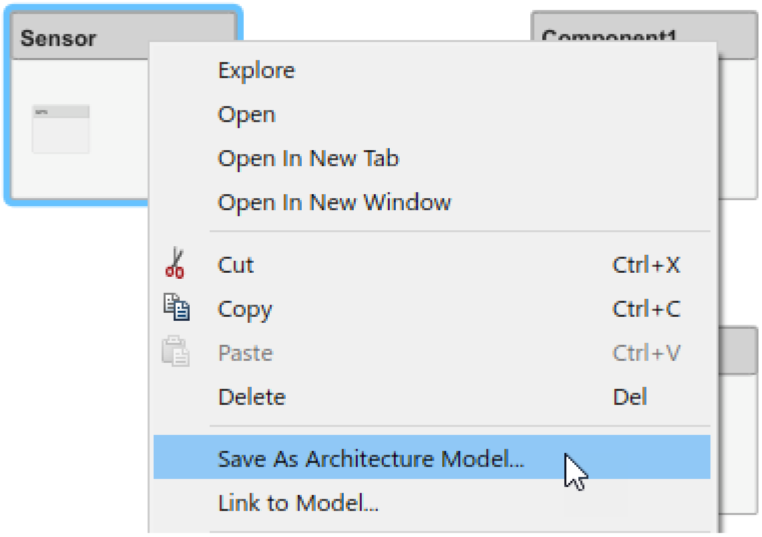
As a result, the new component has the selected components, their ports, and connections as part of its architecture. Any unconnected ports and connections to components outside of the selection become ports on the new component.

Any component that has its own architecture displays a preview of its contents.

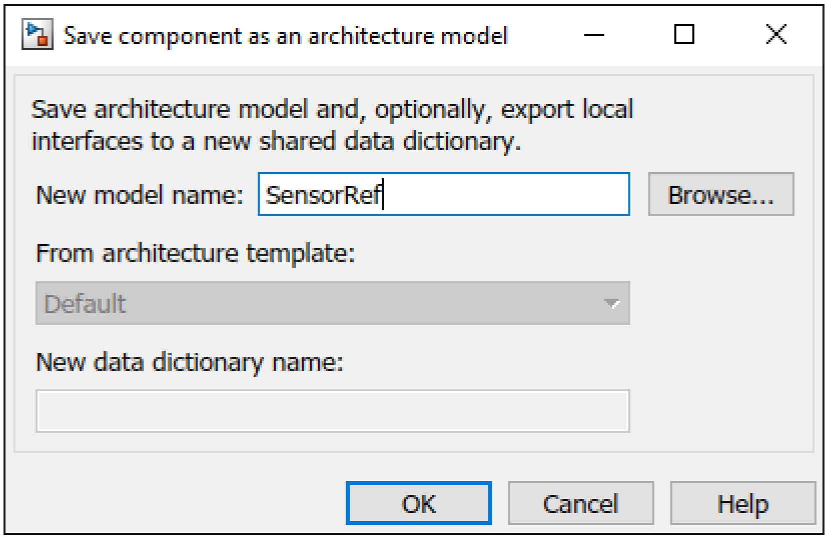
Create a Reference Architecture

Some projects use the same, detailed component in multiple places, and require the design of such a component to be tightly managed. You can create a reference architecture to reuse the architectural definition of a component in the same architecture model or across several architecture models. Create such a reference architecture using this procedure:

1 Right-click the component and select **Save as Architecture Model**.



2 Provide a name for the model. By default, the reference architecture is saved in the same folder as the architecture model. Browse for or type the full path if you want to save it in a different folder.


Save component as an architecture model

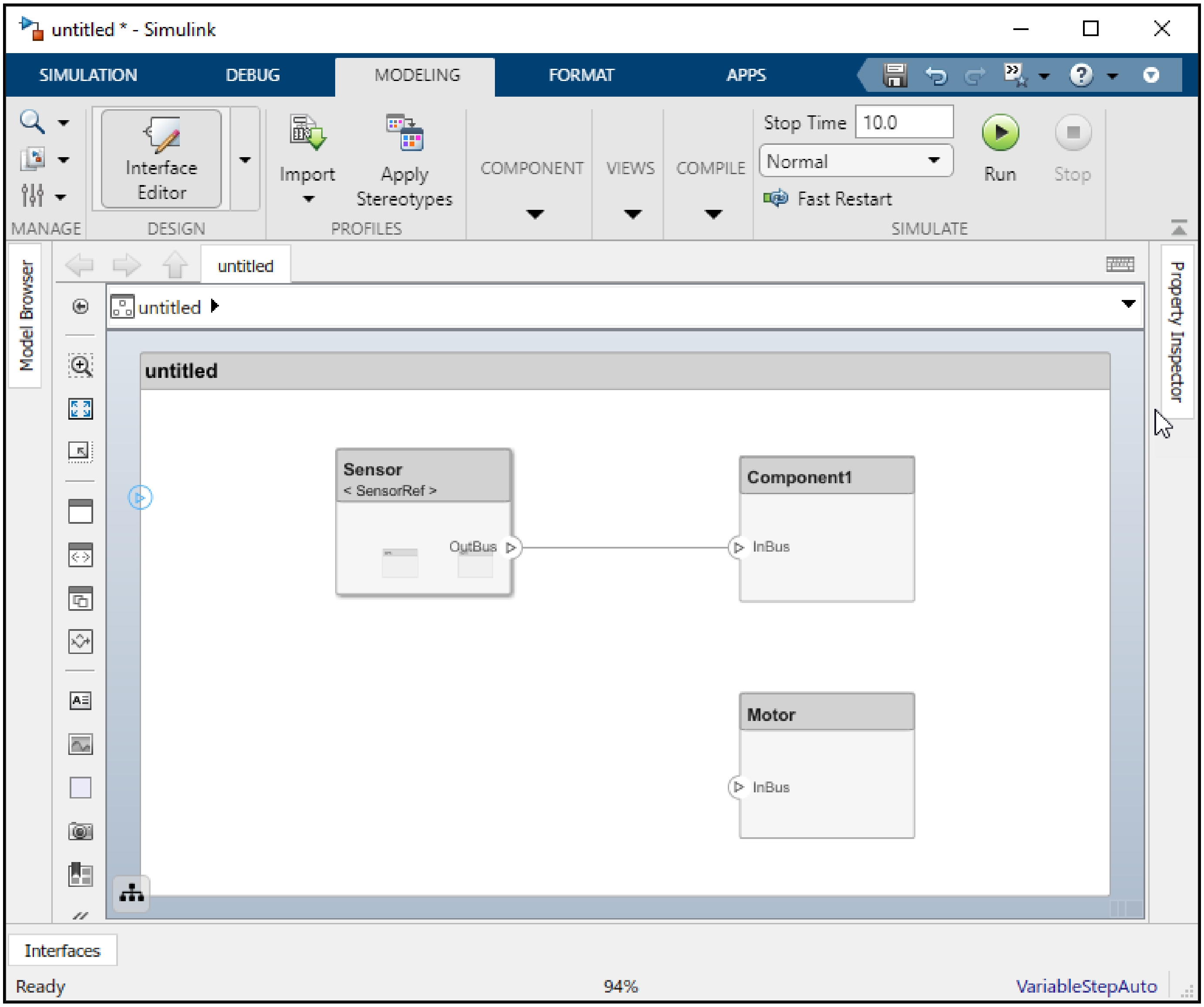
Save architecture model and, optionally, export local interfaces to a new shared data dictionary.

New model name: SensorRef

From architecture template: Default

New data dictionary name:

System Composer creates an architecture model with the provided name, and links the component to the new model. The linked model is indicated in the name of the component between the <> signs.

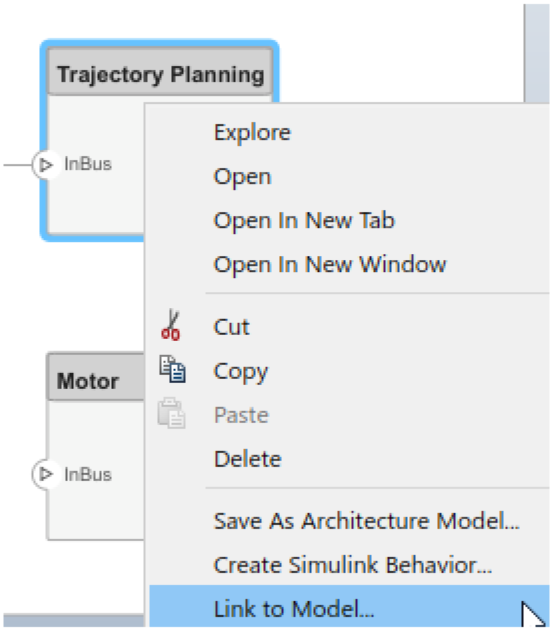


All architecture models can reference this new architecture model through linked components.

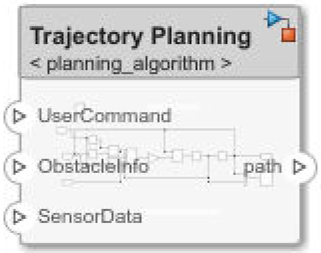
Use a Reference Architecture

You can use a reference architecture, saved in a separate file, by linking to it from a component. Right-click the component and select **Link to Model**. You can also use the **Create Reference** option in the element palette directly to create a component that uses a reference architecture.

To link a selected component to an existing architecture model, right-click the component and select **Link to Model**.



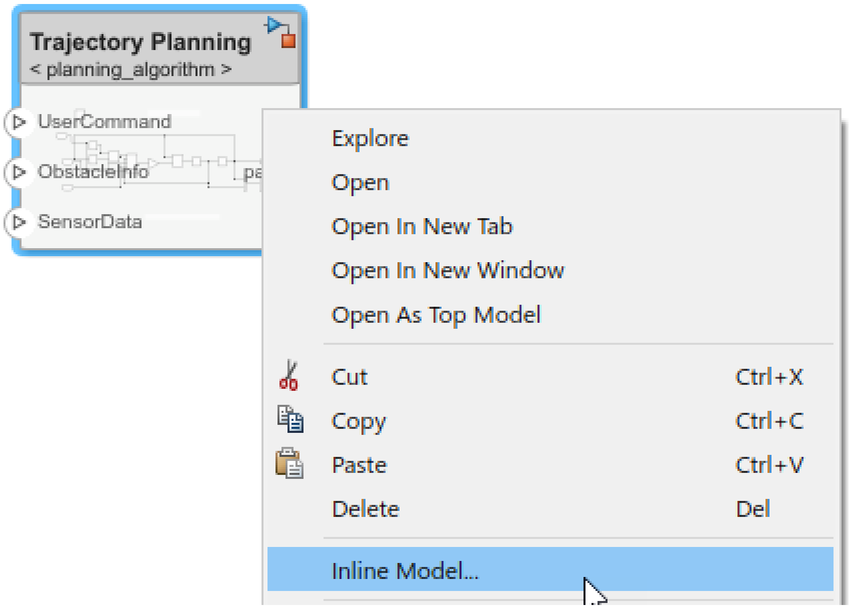
Provide the full path to the reference architecture. If the linked component has its own ports and components, this content is deleted during linking and replaced by that of the reference architecture. The ports of the linked component become the architecture ports in the reference architecture.



Any change made in a reference architecture is immediately reflected in the models that link to it. If you move or rename the reference architecture, the link becomes invalid and the linked component displays an error. Link the component to a valid reference architecture.

Inline a Reference Architecture

In some cases, you have to deviate from the reference architecture for a single component. For example, a comprehensive sensor model, referenced from a local component, may include too many features for the motion control architecture at hand and require simplification for that architecture only. In this case, you can inline the reference architecture to make local changes possible. Right-click a linked component and select **Inline Model**.



This operation provides two options:

- Interface and subcomponents — Ports, interfaces, and subcomponents of the reference architecture are copied to the component.
- Interface only — The ports and designated interfaces of the reference architecture are reflected on the component, but the composition is blank.

Once the reference architecture is inlined, you can start making changes without affecting other architectures. However, you cannot propagate local changes to the reference architecture. If you link to the reference architecture again, local changes are lost.

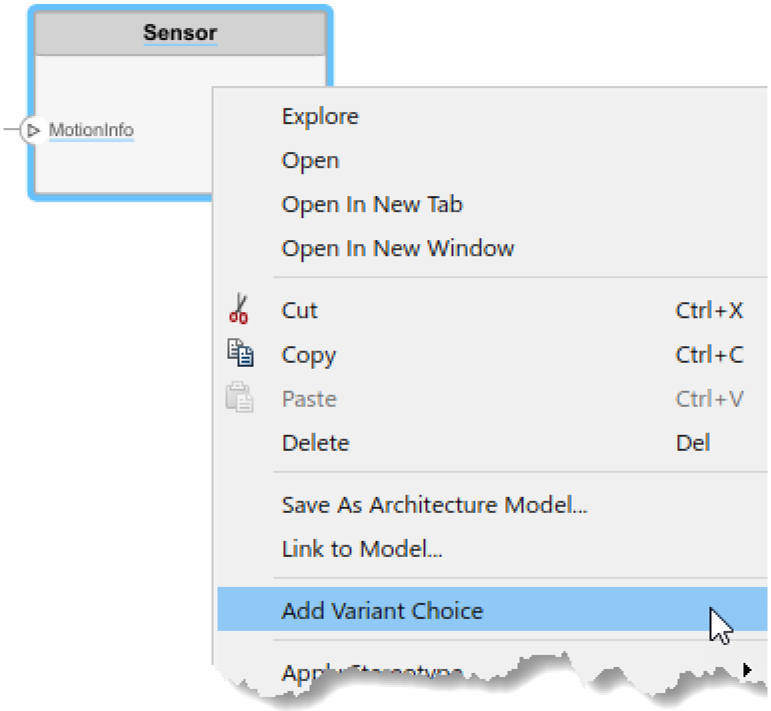
To inline a Stateflow® Chart behavior, see “Inline Stateflow Chart Behavior” on page 5-10.

Create Variants

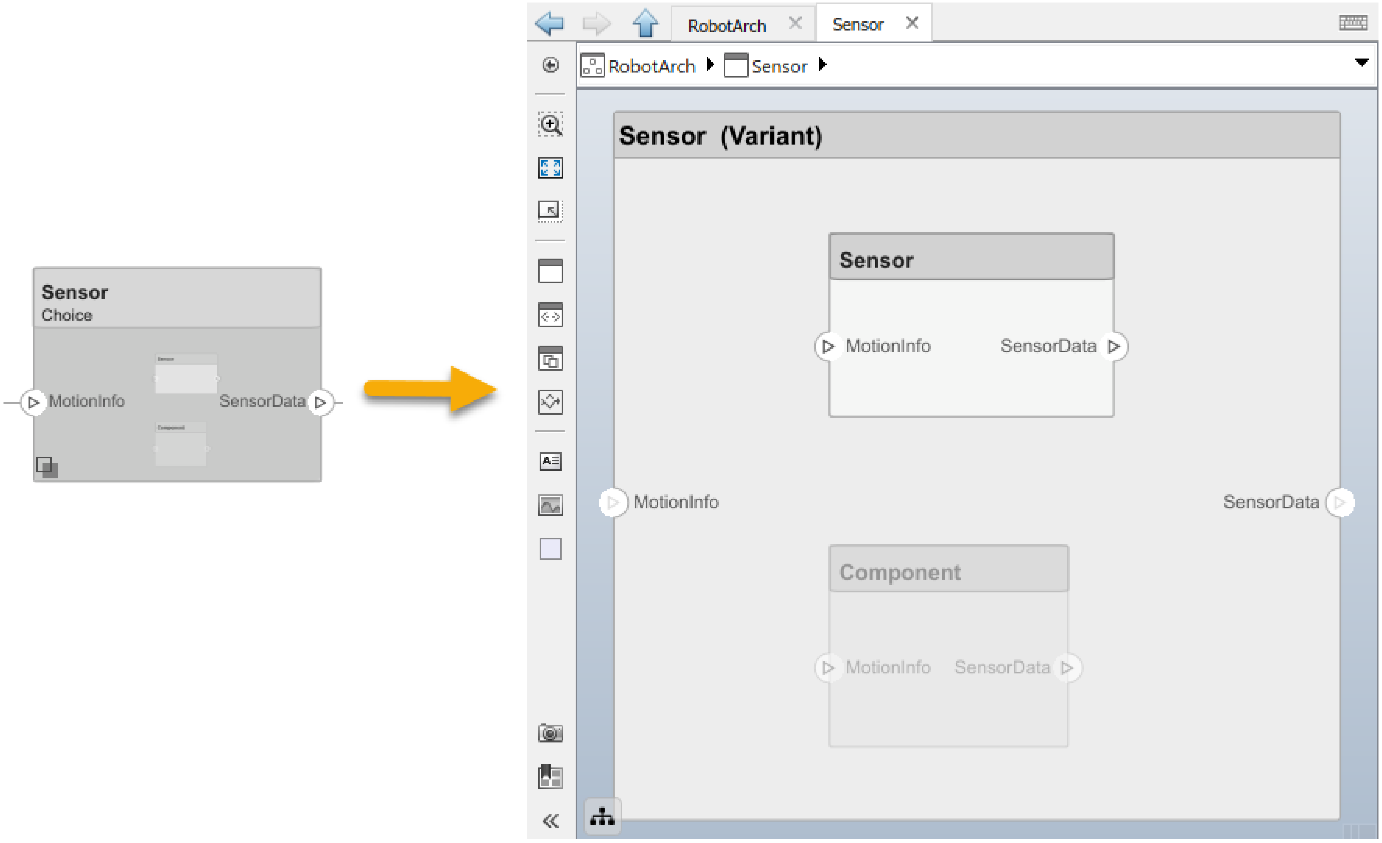
A component can have multiple design alternatives, or variants. A variant is one of many structural or behavioral choices in a variant component. Use variants to quickly swap different architectural designs for a component while performing analysis. A variant control is a string that controls the active variant choice. Set the variant control to programmatically control which variant is active.

You can model variations for any component in a single architecture model. You can define a mix of behaviors (defined in a Simulink model) and architectures (defined in a System Composer architecture model) as variant choices. For example, a component may have two variant options that represent two alternate structural decompositions.

Add variants to a component. Right-click the component and select **Add Variant Choice**.



The ⧉ badge on the component indicates that it is a variant, and a variant choice is added to the existing composition. Double-click the component to see variant choices.

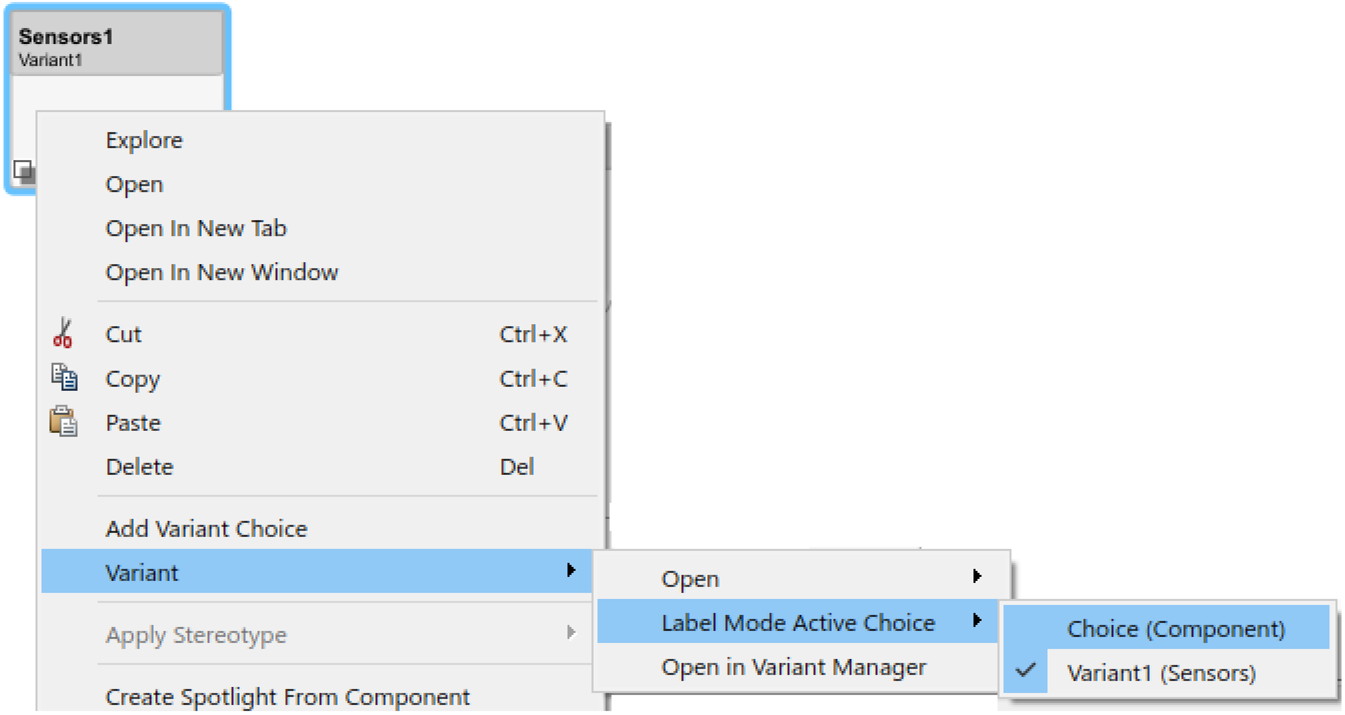


You can add more variant choices to a variant component using the **Add Variant Choice** option.

Open and edit the variant by right-clicking and selecting **Variant > Open > <variant_name>** from the component context menu.

You can also designate a component as a variant upon creation using the ⧉ object in the toolstrip. This creates two variant choices by default.

Activate a specific variant choice using the context menu of the block. Right-click and select **Variant > Label Mode Active Choice > <variant_name>**. The active choice is displayed in the header of the block.



See Also

Functions
`addChoice` | `addVariantComponent` | `inlineComponent` | `linkToModel` | `makeVariant` | `saveAsModel` | `setActiveChoice`

Blocks
Reference Component | Variant Component

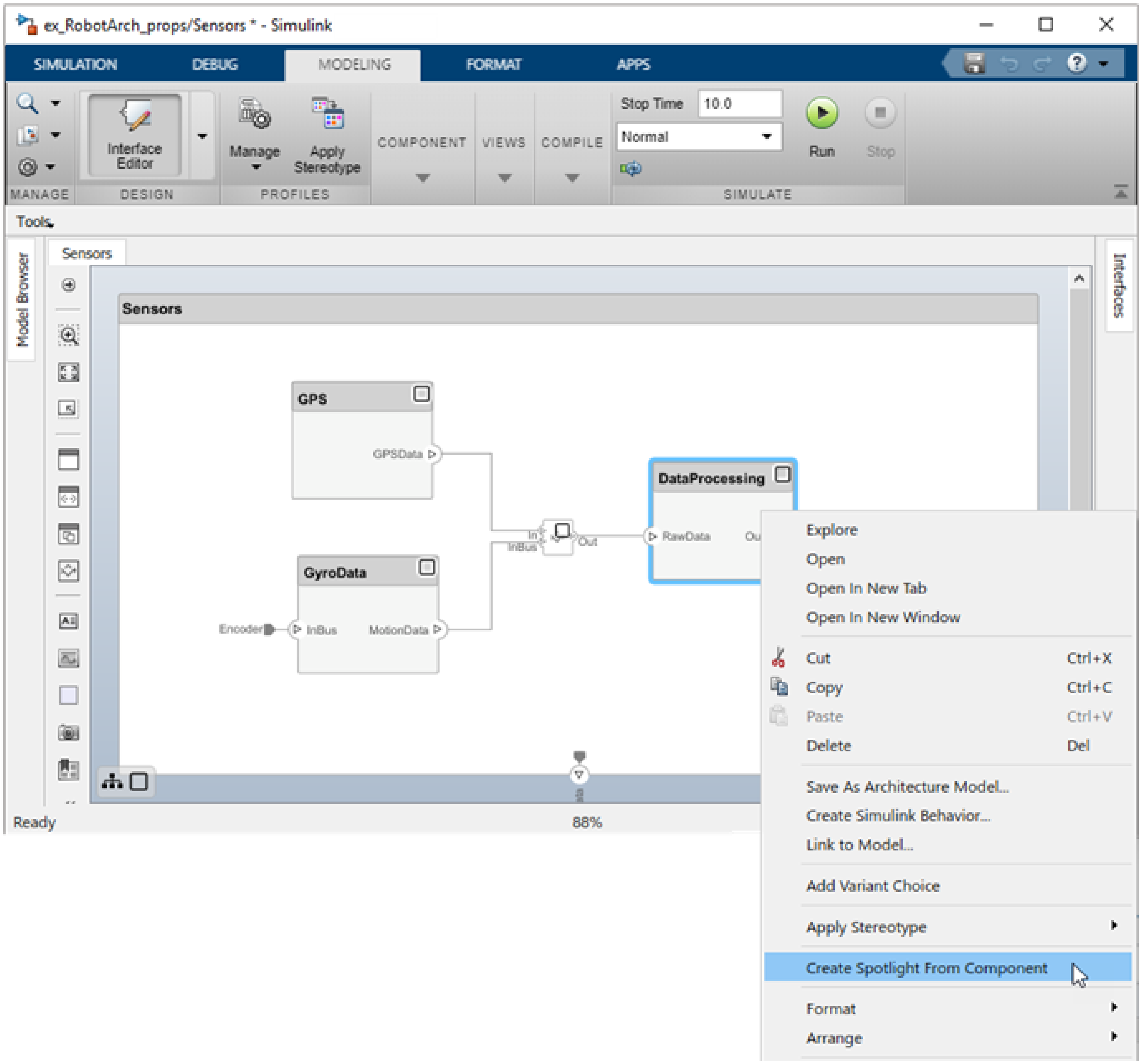
More About

- “Create a Simulink Behavior Model” on page 5-2
- “Link to an Existing Simulink Behavior Model” on page 5-4
- “Create Spotlight Views” on page 1-26

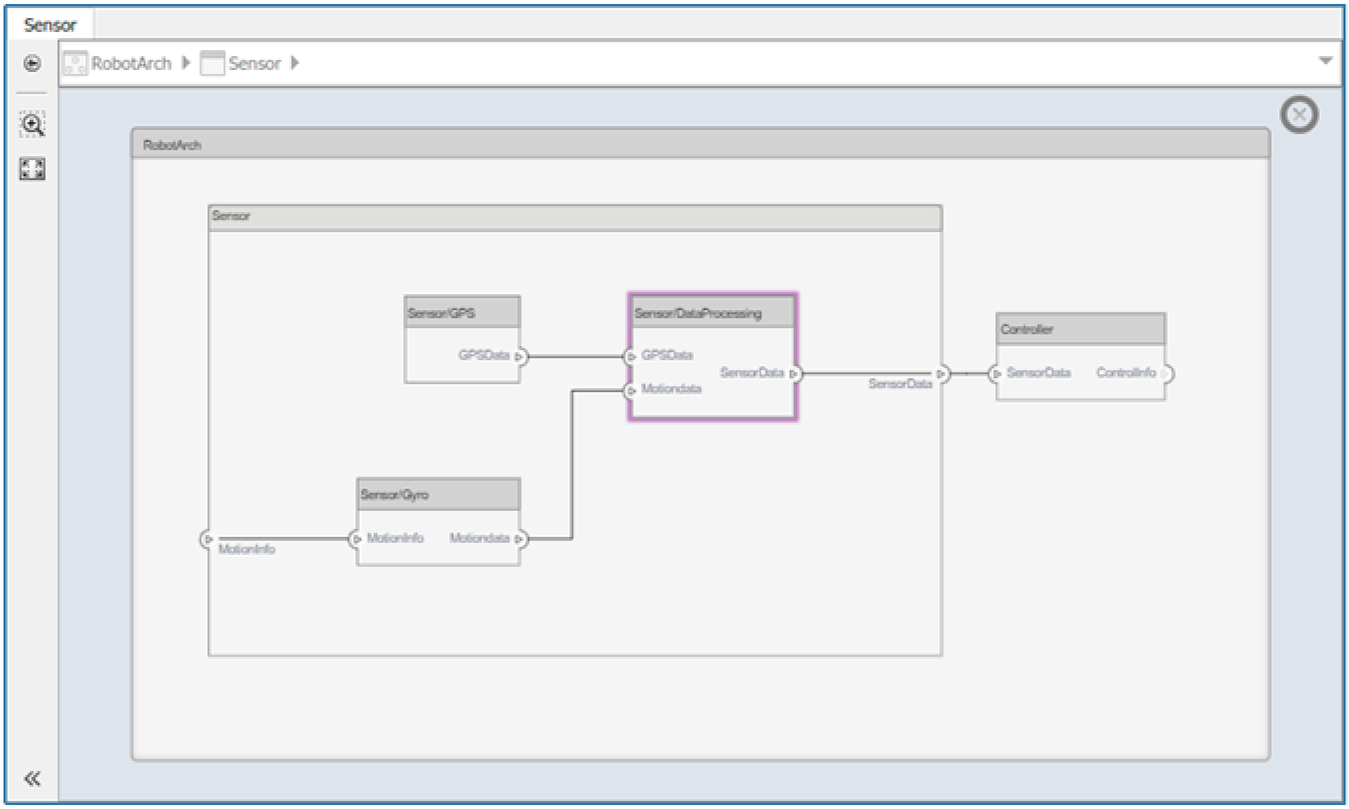
Create Spotlight Views

Any system being designed for a real application is usually very large and complex. It typically consists of many complex functions working together to fulfill the system requirements. In the process of designing and analyzing such architectures, you must understand existing components and what needs to be added. A spotlight view is a simplified view of a model that captures the upstream and downstream dependencies of a specific component of interest.

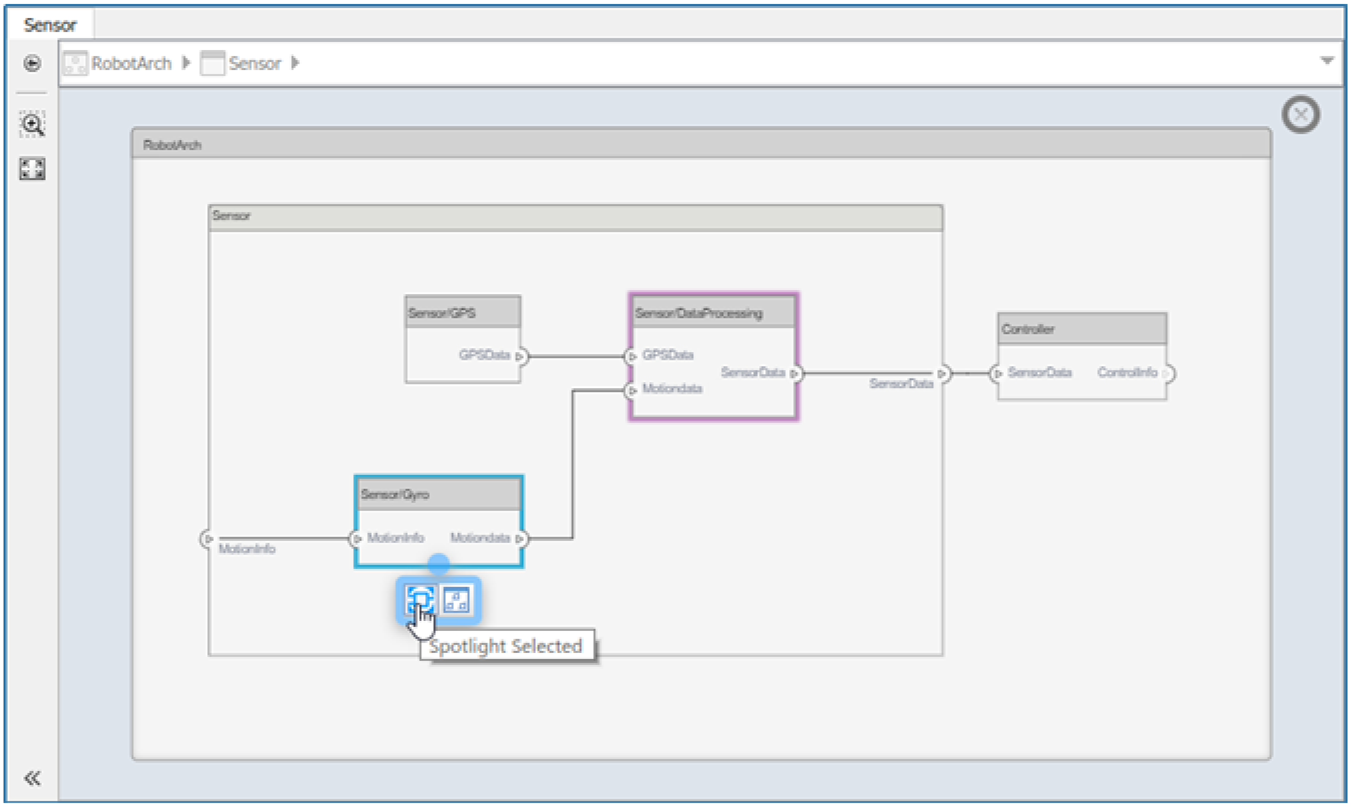
To create a spotlight from the composition, select the component of interest in the canvas, right-click and select **Create Spotlight from Component** either from the **Architecture** menu or the context menu.



The spotlight view launches and shows all model elements to which the component connects in a transparent hierarchy. The spotlight diagram is laid out automatically and cannot be edited.



While in the spotlight view, you can put another component in the spotlight. Select the component and click .



You can make the hierarchy and connectivity of a component visible at all times during model development by opening the spotlight view in a separate window. Show the spotlight view in a dedicated window by first selecting **Open in New Window** in the component context menu and then creating the Spotlight view. Spotlight views are dynamic. Any change in the composition refreshes any open spotlight views. Spotlight views are transient—they are not saved with the model.

You can return to the architecture model view by clicking the ⊗ icon. To view the architecture at the level of a particular component, select the component and click the ⊞ icon.

Note More sophisticated filtering conditions can be created using the Architecture Views Gallery. For details, see “Create Architecture Views Interactively” on page 1-37.

See Also

More About

- “Compose Architecture Visually” on page 1-2
- “Decompose and Reuse Components” on page 1-16

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.

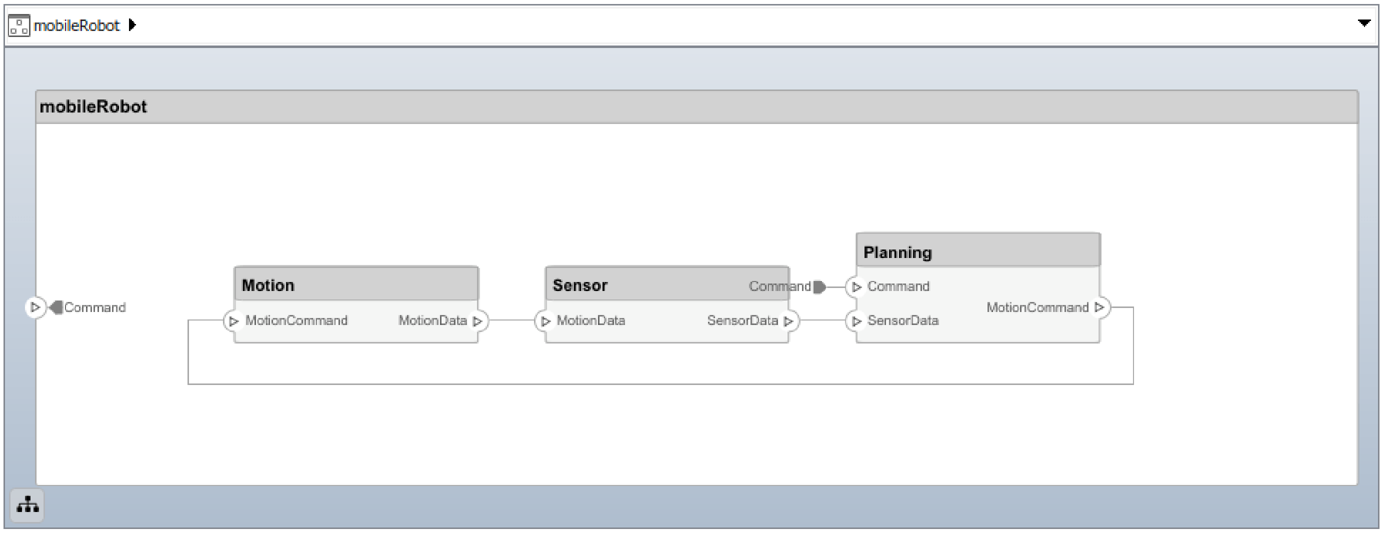
```
save(model)
```

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.

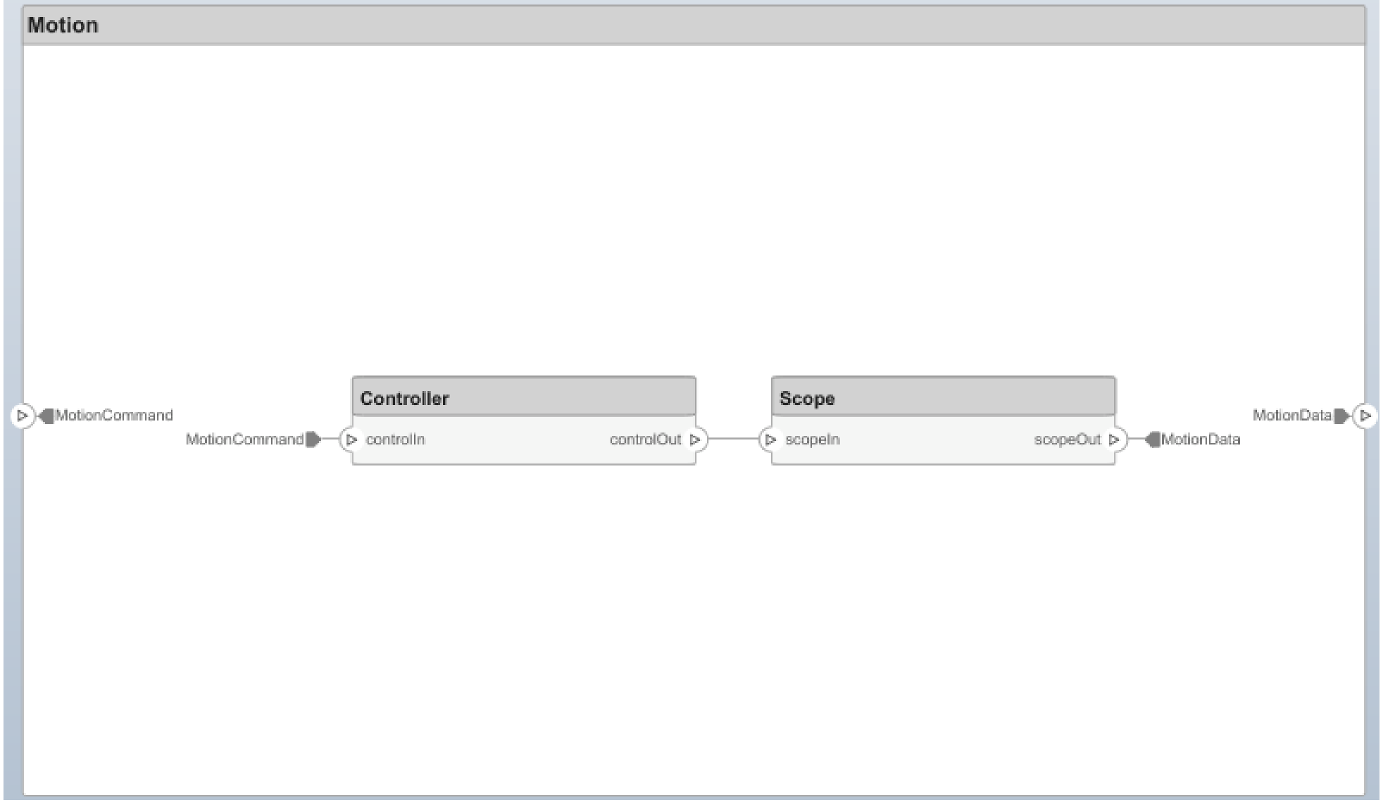
```
motionArch = components(3).Architecture;
motion = motionArch.addComponent({'Controller','Scope'});

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');
```

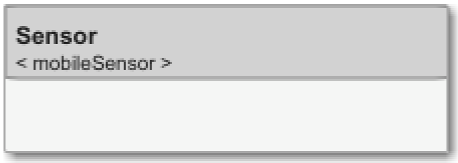


Create a Model Reference

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`.

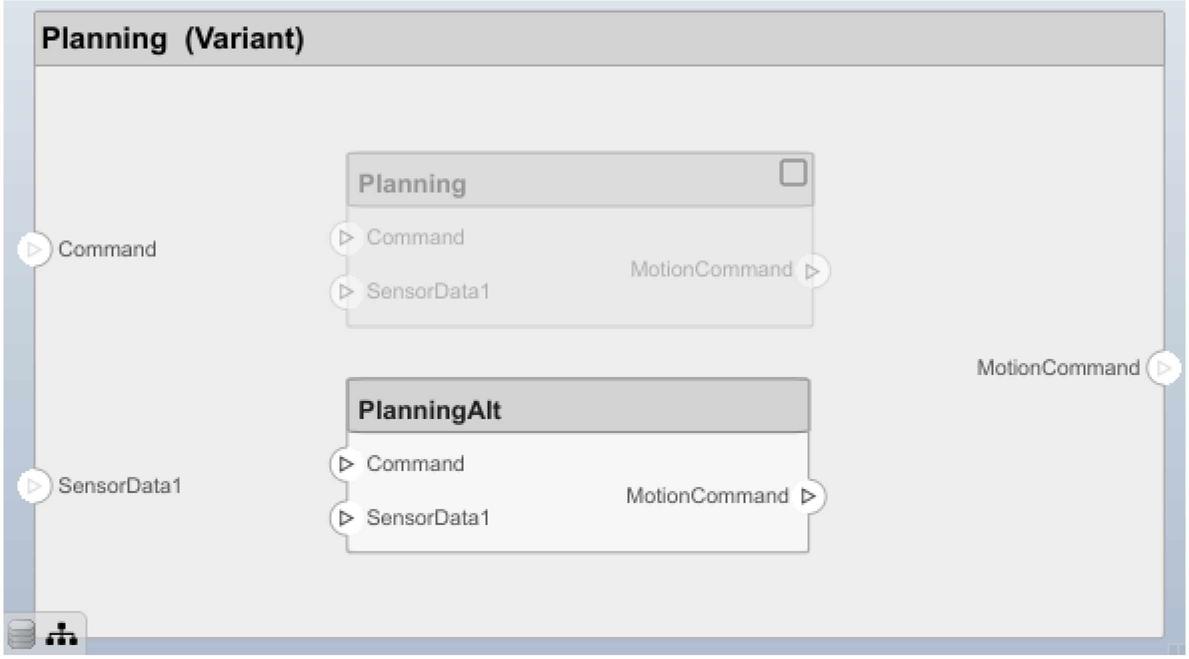
```
setActiveChoice(variantComp,choice2)
planningAltPorts = addPort(choice2.Architecture,{'Command','SensorData1','MotionCommand'},{'in',
planningAltPorts(2).setInterface(interface);
```

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')
```

See Also

Functions
`addChoice` | `addComponent` | `addElement` | `addInterface` | `addPort` | `addProperty` | `addStereotype` | `applyProfile` | `applyStereotype` | `batchApplyStereotype` | `closeAll` | `connect` | `createDictionary` | `createModel` | `createProfile` | `getPort` | `linkDictionary` | `linkToModel` | `makeVariant` | `save` | `save` | `setActiveChoice` | `setInterface` | `setProperty`

Blocks
Component | Reference Component | Variant Component

More About

- “Compose Architecture Visually” on page 1-2
- “Define Profiles and Stereotypes” on page 4-2
- “Use Stereotypes and Profiles” on page 4-10

- “Assign Interfaces to Ports” on page 3-7
- “Decompose and Reuse Components” on page 1-16

Create Architecture Views Interactively

The structural hierarchy of a system typically differs from the hierarchy of the system's functional requirements. With architecture views, you can view a system based on different hierarchies.

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.

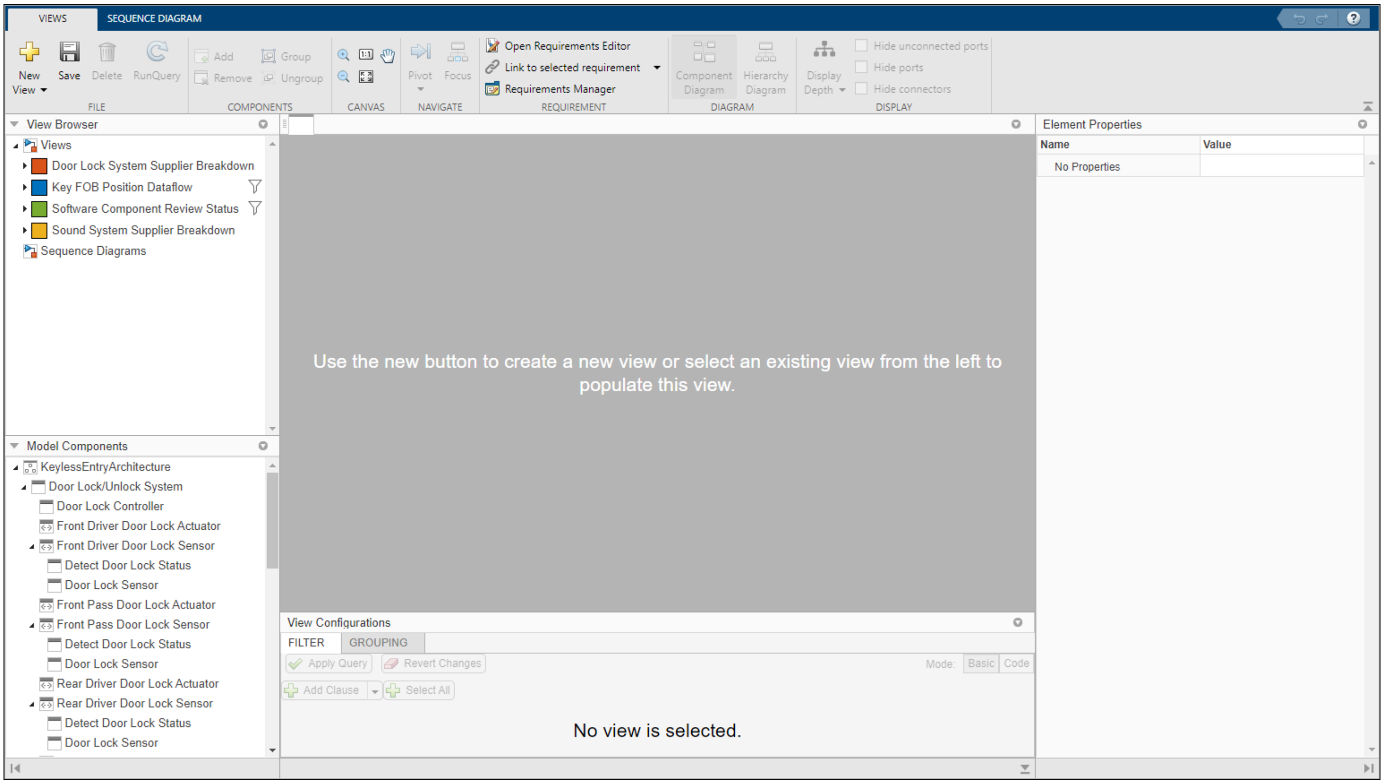
A viewpoint represents a stakeholder perspective that specifies the contents of the view.

For example, you can author a system using the requirements. This view allows you to better understand what components you need to satisfy your requirements while not necessarily focusing on the structure.

You can create an architecture view interactively with automation or construct them manually. This example uses the architecture model for a keyless entry system, `scKeylessEntrySystem`, to create views.

Create Filtered Views with Grouping Criteria

1. In the MATLAB Command Window, enter `scKeylessEntrySystem`. The architecture model opens in the Simulink Editor.
2. In the **Views** section, click **Architecture Views** to open the Architecture Views Gallery.

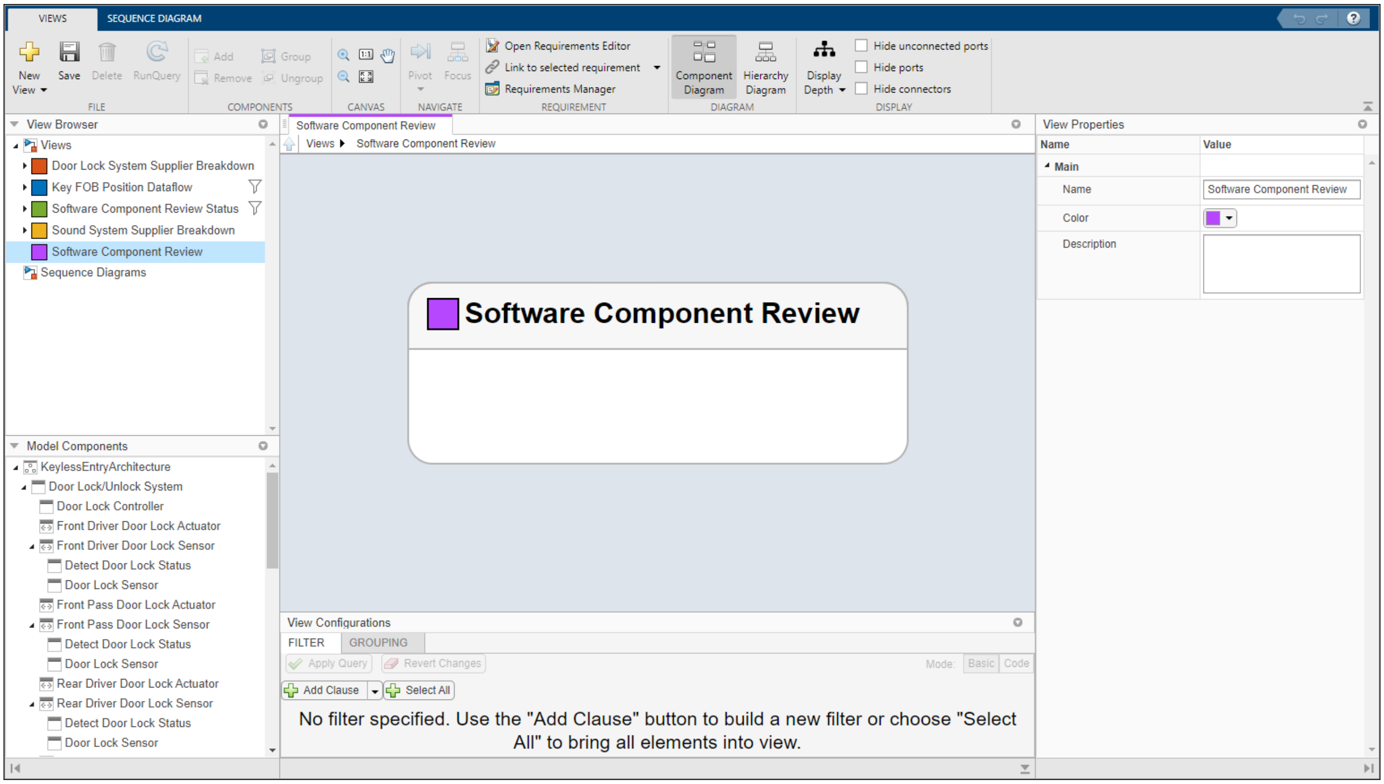


Use the new button to create a new view or select an existing view from the left to populate this view.

No view is selected.

3 Click **New View** to create a new view.

4 In **View Properties** on the right pane, in the **Name** box, enter a name for this view, for example, `Software Component Review`. Choose a **Color** and enter a **Description**, if necessary.

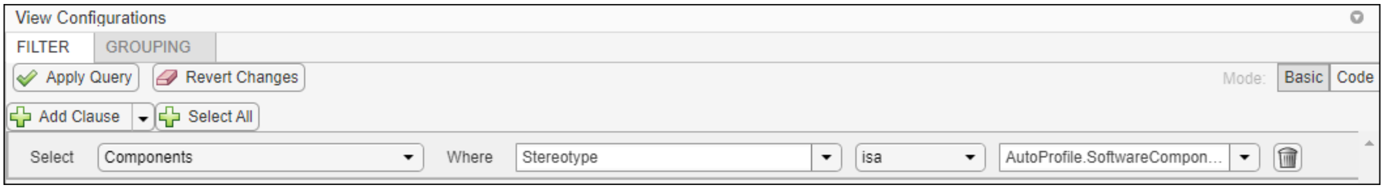


Name: Software Component Review

No filter specified. Use the "Add Clause" button to build a new filter or choose "Select All" to bring all elements into view.

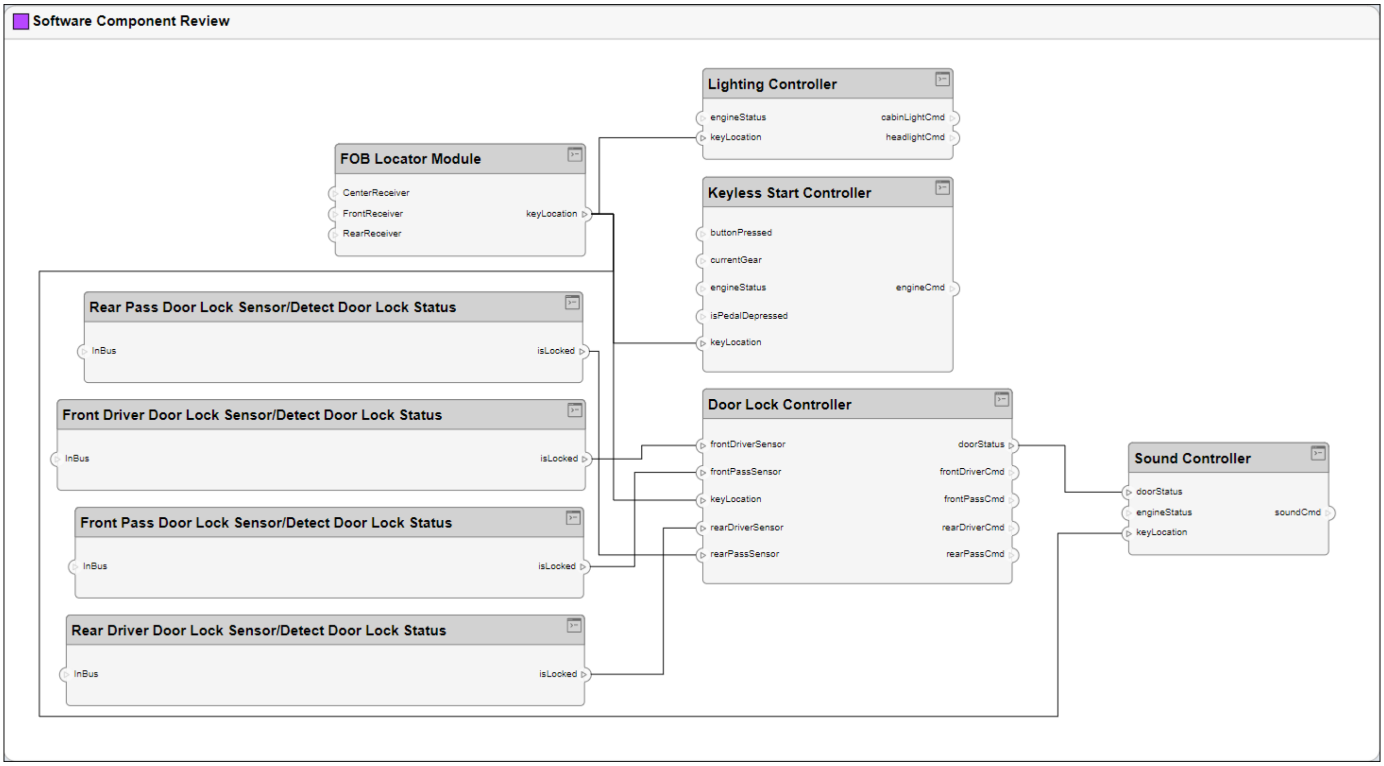
5 In the **View Configurations** pane, select **Filter** to add a new form-based criterion to the filter.

6 Select **Add Clause**. From the **Select** drop-down, select `Components`. From the **Where** drop-down, select `Stereotype`. In the text box, select `AutoProfile.SoftwareComponent` from the drop-down.



Select Components Where Stereotype isa AutoProfile.SoftwareCompon...

7 Click **Apply Query**. An architecture view is created using the query from the **Filter** box. The view is filtered to select all components with the `SoftwareComponent` stereotype applied to them.



8 In the View Configurations pane, select **Grouping**.

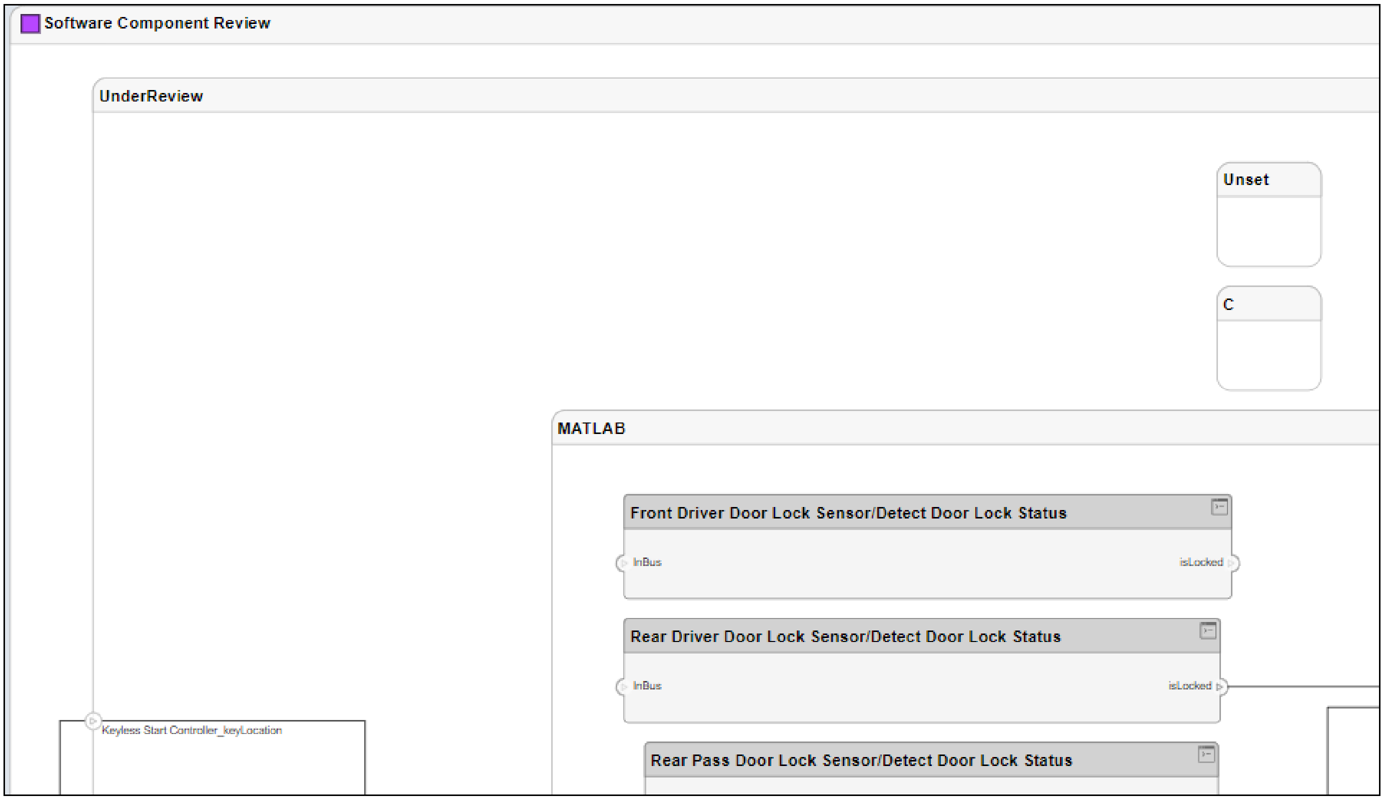
9 To choose a property enumeration for grouping, click **Add Group By**.

10 Select `AutoProfile.BaseComponent.ReviewStatus` from the drop-down.

11 Click **Add Group By** again.

12 Select `AutoProfile.SoftwareComponent.ImplementationLanguage` from the drop-down.

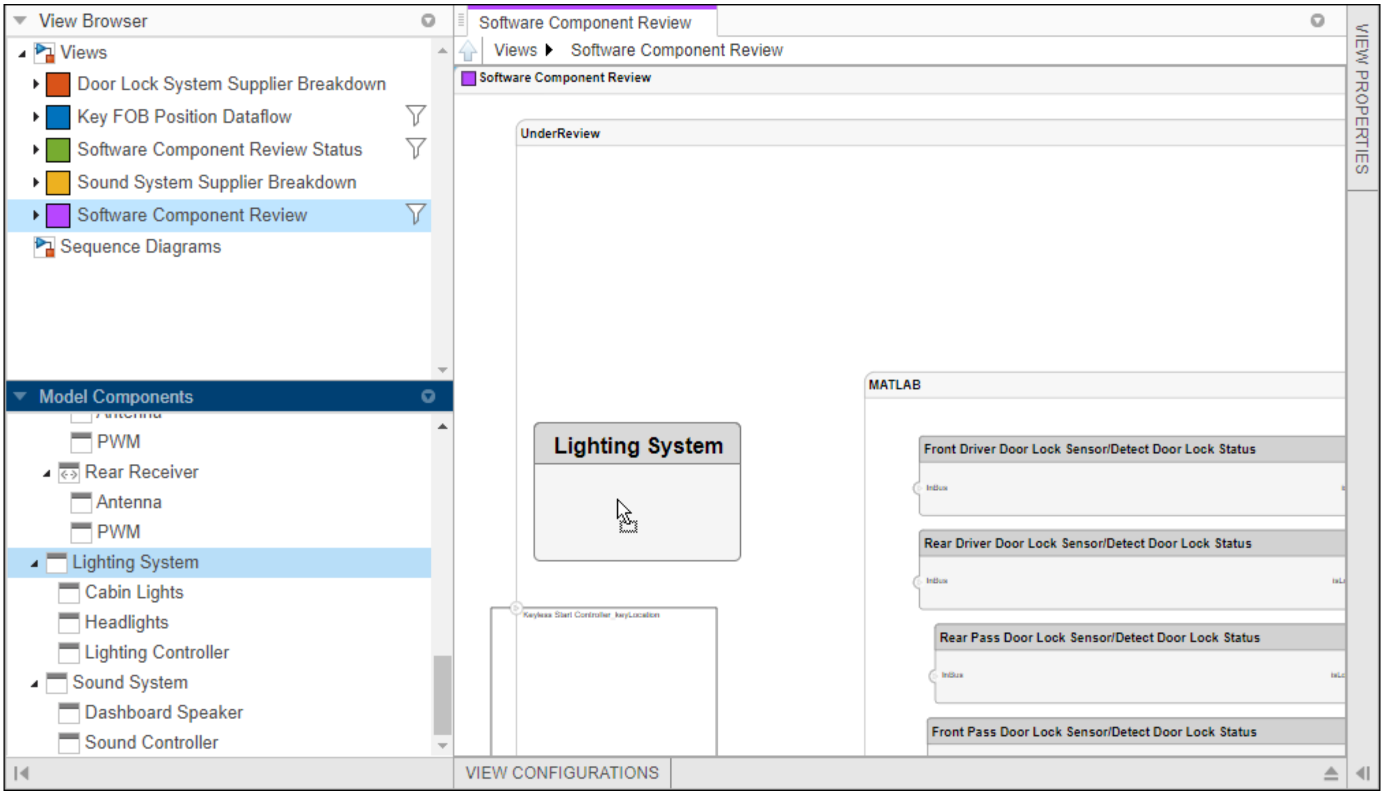
13 Click **Apply Query**.


Software Component Review
UnderReview
Unset
C
MATLAB
Front Driver Door Lock Sensor/Detect Door Lock Status
InBus
isLocked
Rear Driver Door Lock Sensor/Detect Door Lock Status
InBus
isLocked
Rear Pass Door Lock Sensor/Detect Door Lock Status
Keyless Start Controller_keyLocation

Interactively Add and Remove Elements from Views

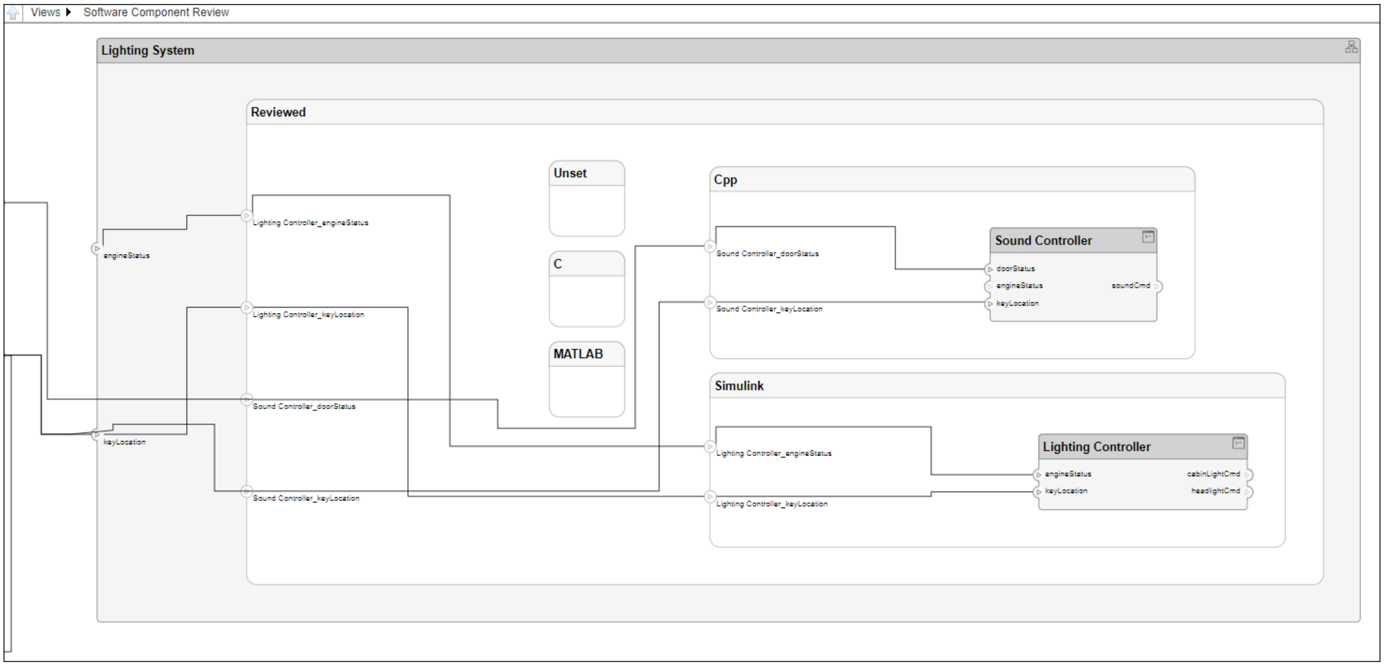
1 To add more components to the view, drag and drop components from **Model Components**. Drag and drop the `Lighting System` component to the `Software Component Review` view. Alternatively, use the **Add** button on the toolstrip. You can also use the keyboard shortcut **Ctrl+I** to add component instantiations to your view when they are selected.

Note Interactively adding and removing elements from your view with an associated query is not supported. You will receive a warning message: Remove associated query? Press **OK** to proceed.


View Browser
Views
Door Lock System Supplier Breakdown
Key FOB Position Dataflow
Software Component Review Status
Sound System Supplier Breakdown
Software Component Review
Sequence Diagrams
Model Components
Antenna
PWM
Rear Receiver
Antenna
PWM
Lighting System
Cabin Lights
Headlights
Lighting Controller
Sound System
Dashboard Speaker
Sound Controller
Software Component Review
Views ▸ Software Component Review
Software Component Review
UnderReview
MATLAB
Lighting System
Front Driver Door Lock Sensor/Detect Door Lock Status
Rear Driver Door Lock Sensor/Detect Door Lock Status
Rear Pass Door Lock Sensor/Detect Door Lock Status
Front Pass Door Lock Sensor/Detect Door Lock Status
InBus
Keyless Start Controller_keyLocation
VIEW CONFIGURATIONS
VIEW PROPERTIES

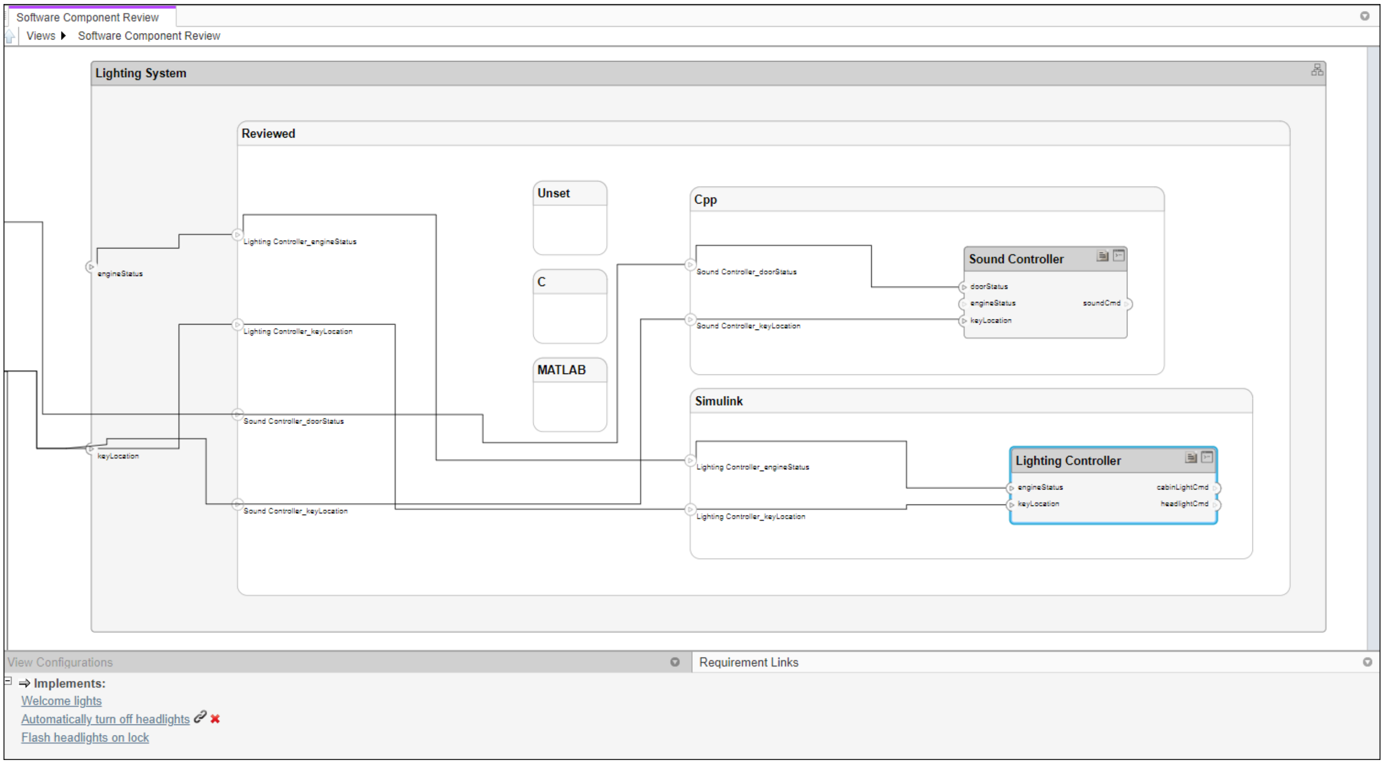
You can use the keyboard shortcut **Delete** to delete components from the view.

2 Observe that `Lighting System` has been added to the view.



3 From the **Requirement** menu, select **Requirements Manager**. The **Requirement Links** tab appears at the bottom of the `Software Component Review` view.

4 Select the Lighting Controller component and see the linked requirement Automatically turn off headlights.


Software Component Review
Views ▸ Software Component Review
Lighting System
Reviewed
Unset
C
MATLAB
Cpp
Simulink
engineStatus
keyLocation
Lighting Controller_engineStatus
Lighting Controller_keyLocation
Sound Controller_doorStatus
Sound Controller_keyLocation
Sound Controller_doorStatus
Sound Controller_keyLocation
Sound Controller
doorStatus
engineStatus
keyLocation
soundCmd
Lighting Controller_engineStatus
Lighting Controller_keyLocation
Lighting Controller
engineStatus
keyLocation
cabinLightCmd
headlightCmd
View Configurations
Requirement Links
Implements:
Welcome lights
Automatically turn off headlights
Flash headlights on lock

5 Select the requirement Automatically turn off headlights to open the Requirement Editor to view or modify requirement links.

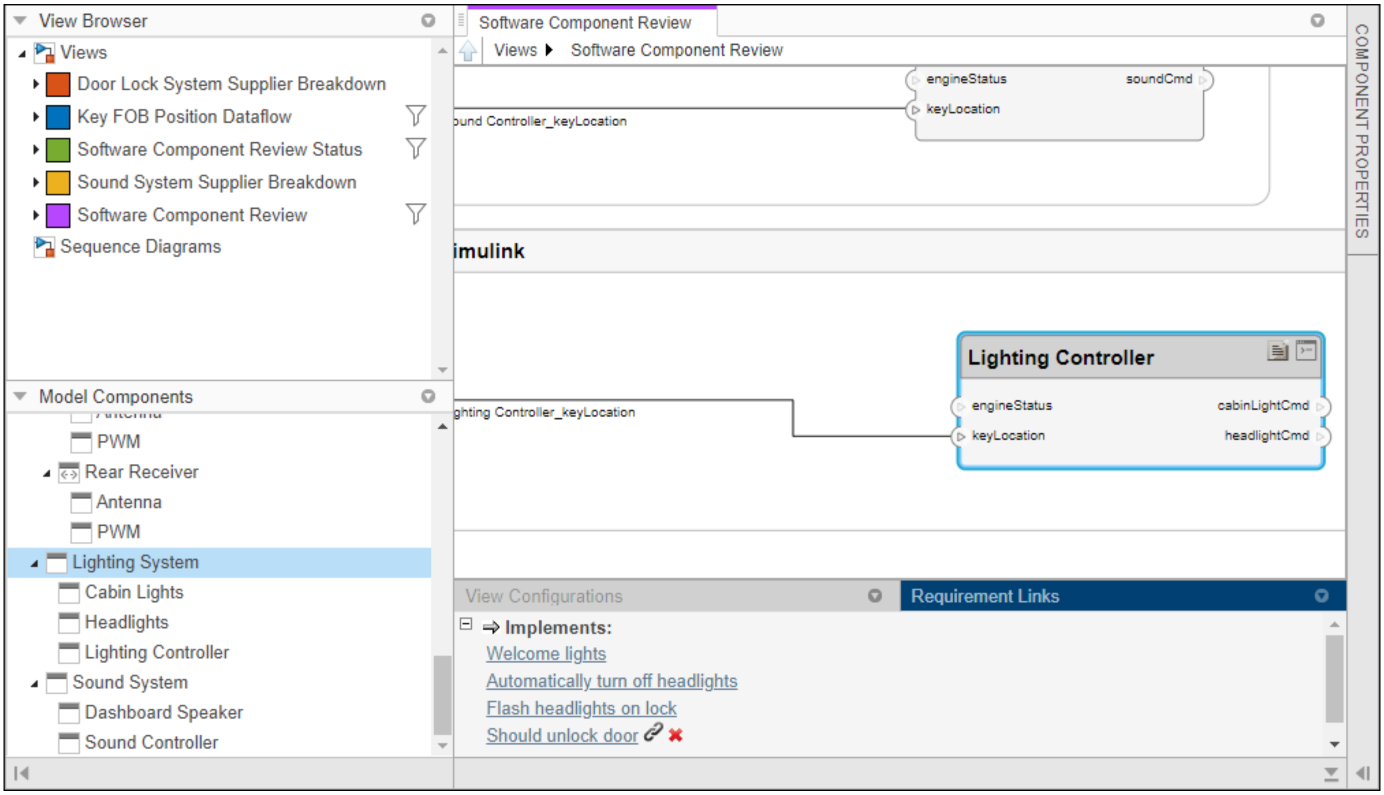
Add or Remove Requirements Links from Views

1 In the Architecture Views Gallery, from the **Requirement** menu, select **Open Requirements Editor** if the Requirement Editor is not open already.

2 Select the Should unlock door requirement.

3 Return to the Architecture Views Gallery. In the Software Component Review view select the Lighting Controller component.

4 From the **Requirement** menu, select **Link to selected requirement**. The new requirement Should unlock door is added.



5 To remove a requirement link, select ✖ and confirm deletion.

See Also

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

More About

- "Create Architectural Views Programmatically" on page 1-45
- "Display Component Hierarchy Using Hierarchy Views" on page 1-58

Create Architectural Views Programmatically

You can create an architecture view programmatically. This topic presents two examples of creating architecture views programmatically and shows you how to use queries to find elements in a System Composer model.

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.

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'))
```

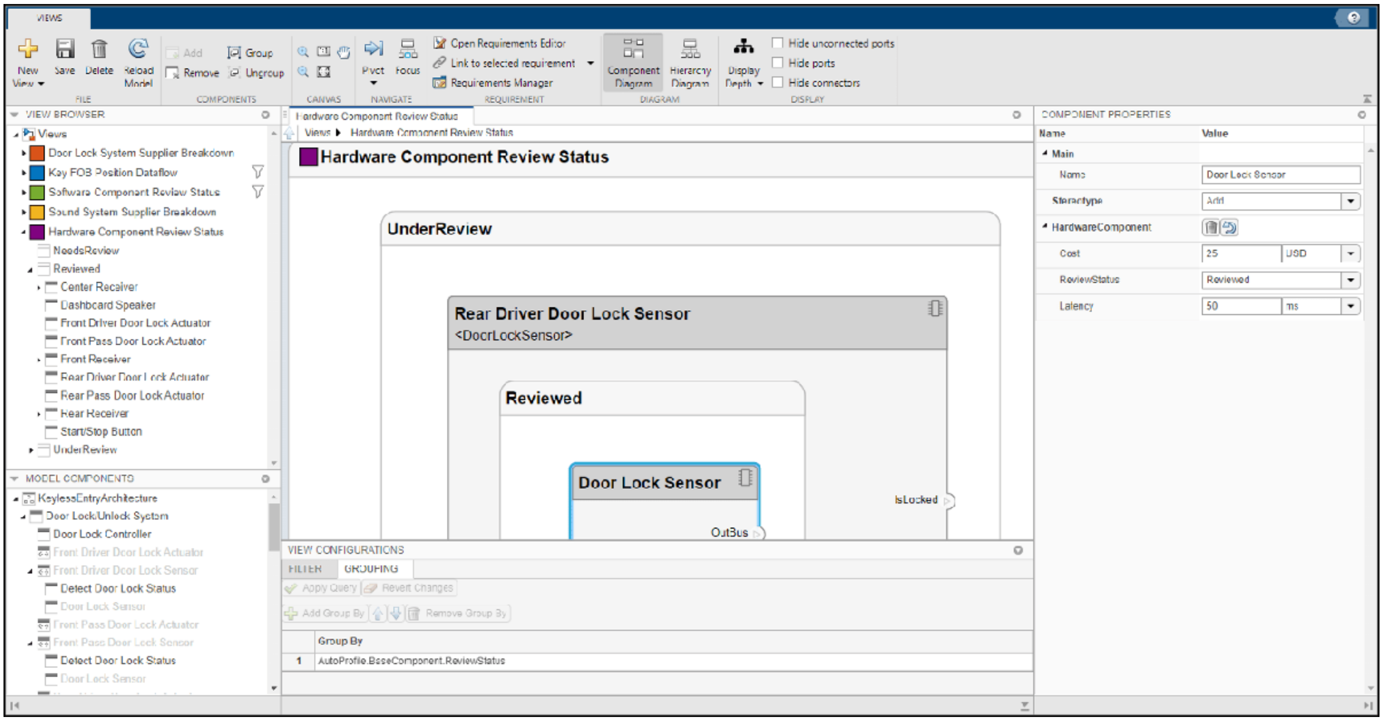
```
hwCompQuery = 
  HasStereotype with properties:

    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
```



Group By
1 AutoProfile.BaseComponent.ReviewStatus

Name	Value
Main	
Name	Door Lock Sensor
Stereotype	Add
HardwareComponent	
Cost	25 USD
ReviewStatus	Reviewed
Latency	50 ms

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.

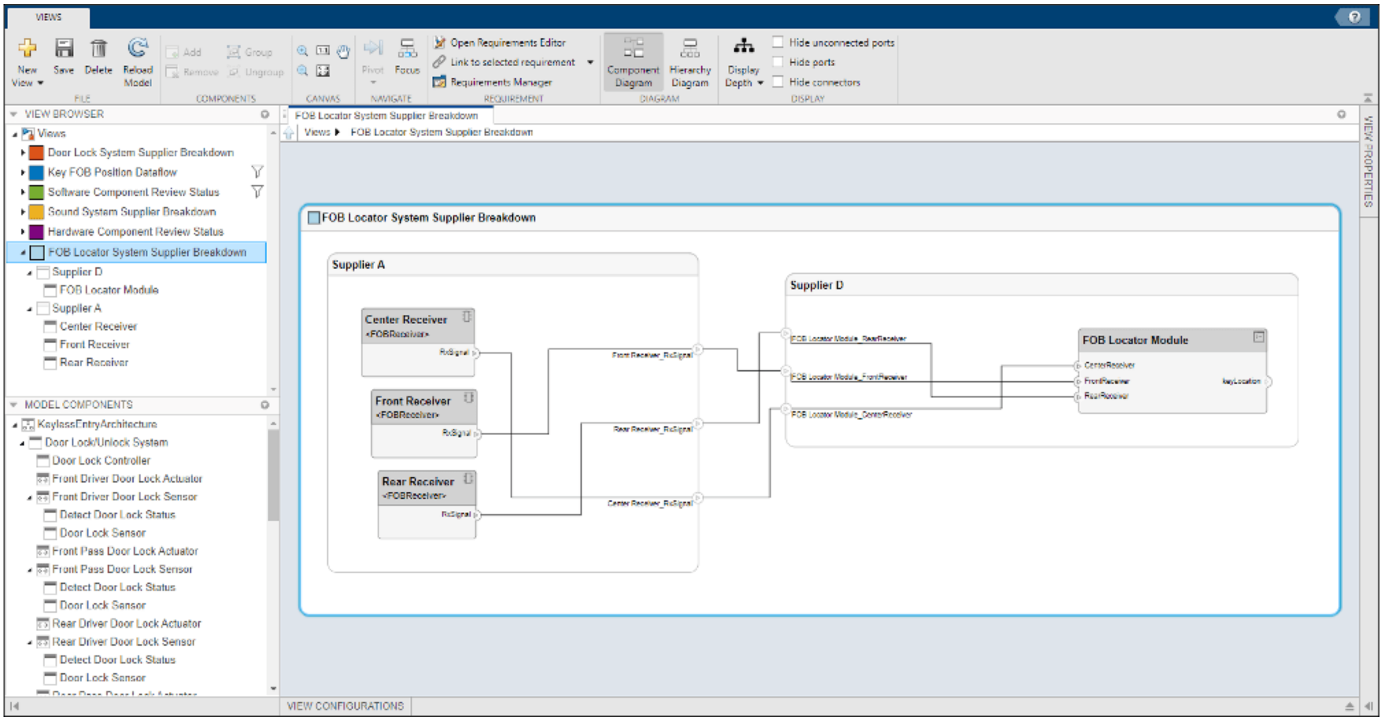
```
supplierD = fobSupplierView.Root.createSubGroup('Supplier D');
supplierD.addElement('KeylessEntryArchitecture/FOB Locator System/FOB Locator Module');
```

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)
```



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;
```

See Also

`createView` | `deleteView` | `find` | `getView` | `lookup` | `openViews` | `systemcomposer.query.Constraint` | `systemcomposer.view.ElementGroup` | `systemcomposer.view.View`

More About

- “Create Architecture Views Interactively” on page 1-37
- “Display Component Hierarchy Using Hierarchy Views” on page 1-58

Import and Export Architecture Models

To build a System Composer model, you can import information about components, ports, and connections in a predefined format using MATLAB table objects. You can extend these tables and add information like applied stereotypes, property values, linked model references, variant components, interfaces, and requirement links.

Similarly, you can export information about components, hierarchy of components, ports on components, connections between components, linked model references, variants, stereotypes on elements, interfaces, and requirement links.

Define a Basic Architecture

The minimum required structure for a System Composer model consists of these sets of information:

- Components table
- Ports table
- Connections table

To import additional elements, you need to add columns to the tables and add specific values for these elements.

Components Table

The information about components is passed as values in a MATLAB table against predefined column names, where:

- `Name` is the component name.
- `ID` is a user-defined ID used to map child components and add ports to components.
- `ParentID` is the parent component ID.

For example, `Component_1_1` and `Component_1_2` are children of `Component_1`.

Name	ID	ParentID
root	0	
Component_1	1	0
Component_1_1	2	1
Component_1_2	3	1
Component_2	4	0

Ports Table

The information about ports is passed as values in a MATLAB table against predefined column names, where:

- `Name` is the port name.
- `Direction` is an input or output port direction.
- `ID` is a user-defined port ID used to map ports to port connections.

- `CompID` is the ID of the component to which the port is added. It is the component passed in the components table.

Name	Direction	ID	CompID
`Port1`	`Output`	1	1
`Port2`	`Input`	2	4
`Port1_1`	`Output`	3	2
`Port1_2`	`Input`	4	3

Connections Table

The information about connections is passed as values in a MATLAB table against predefined column names, where:

- `Name` is the connection name.
- `ID` is connection ID used to check that the connections are properly created during the import process.
- `SourcePortID` is the ID of the source port.
- `DestPortID` is the ID of the destination port.

Name	ID	SourcePortID	DestPortID
`Conn1`	1	1	2
`Conn2`	2	3	4

Import a Basic Architecture

Import the basic architecture from the tables created above into System Composer from the MATLAB Command Window.

```
systemcomposer.importModel('importedModel',components,ports,connections)
```

The basic architecture model opens.

Tip The tables do not include information about the model's visual layout. You can arrange the components manually or use **Architecture > Arrange > Arrange Automatically**.

Extend the Basic Architecture Import

You can import other model elements into the basic structure tables.

Import Interfaces and Map Ports to Interfaces

To define the interfaces, add interface names in the `ports` table to associate ports to corresponding `portInterfaces` table. Create a table similar to `components`, `ports`, and `connections`. Information like interface name, associated element name along with data type, dimensions, units, complexity, minimum, and maximum values are passed to the `importModel` function in a table format shown below.

Name	**ID**	**ParentID**	**DataType**	**Dimensions**	**Units**	**Complexity**	**Minimum**	**Maximum**
interface1	1							
elem1	2	interface1	interface3	1	""	real	"[]"	"[]"
interface2	3		1	1	""	real	"[]"	"[]"
elem2	4	interface1	1	1	""	real	"[]"	"[]"

Note Anonymous interfaces cannot be nested. You cannot define an anonymous interface as the data type of elements.

To map the added interface to ports, add column `InterfaceID` in the `ports` table to specify the interface to be linked. For example, `interface1` is mapped to `Port1` as shown below.

Name	**Direction**	**ID**	**CompID**	**InterfaceID**
Port1	Output	1	1	interface1
Port2	Input	2	4	interface2
Port1_1	Output	3	2	""
Port1_2	Input	4	3	interface1

Import Variant Components, Stateflow Behaviors, or Reference Components

You can add variant components just like any other component in the `components` table, except you specify the name of the active variant. Add choices as child components to the variant components. Specify the variant choices as string values in the `VariantControl` column. You can enter expressions in the `VariantCondition` column.

Add a variant component `VarComp` using component type `Variant` with choices `Choice1` and `Choice2`. Set `Choice2` as the active choice.

To add a referenced Simulink model, change the component type to `Behavior` and specify the reference model name `simulink_model`.

To add a Stateflow Chart behavior on a component, change the component type to `StateflowBehavior`. If System Composer does not detect a license or installation of Stateflow, a `Composition` component is imported instead.

Name	**ID**	**ParentID**	**ReferenceModelName**	**ComponentType**	**ActiveChoice**	**VariantControl**	**VariantCondition**
root	0						
Component1	C1	0	simulink_model	Behavior			

Name	ID	ParentID	ReferenceModelName	ComponentType	ActiveChoice	VariantControl	VariantCondition
VarComp	V2	0		Variant	Choice2		
Choice1	C6	V2				petrol	
Choice2	C7	V2				diesel	
Component3	C3	0		Stateflow Behavior			
Component1_1	C4	C1					
Component1_2	C5	C1					

Pass the modified `components` table along with the `ports` and `connections` tables to the `importModel` function.

Apply Stereotypes and Set Property Values on Imported Model

To apply stereotypes on components, ports, and connections, add a `StereotypeNames` column to the `components` table. To set the properties for the stereotypes, add a column with a name defined using the profile name, stereotype name, and property name. For example, name the column `UAVComponent_OnboardElement_Mass` for a `UAVComponent` profile, a `OnBoardElement` stereotype, and a `Mass` property.

You set the property values in the format `value{units}`. Units and values are populated from the default values defined in the loaded profile file.

Name	ID	ParentID	StereotypeNames	UAVComponent_OnboardElement_Mass	UAVComponent_OnboardElement_Power
root	0				
Component_1	1	0	UAVComponent.OnboardElement	0.93{kg}	0.65{mW}
Component_1_1	2	1			
Component_1_2	3	1	UAVComponent.OnboardElement	0.93{kg}	""
Component_2	4	0			

Assign Requirement Links on Imported Model

To assign requirement links to the model, add a `requirementLinks` table with these required columns:

- `Label` is the name of the requirement.
- `ID` is the ID of the requirement.
- `SourceID` is the architecture element to which the requirement is attached.

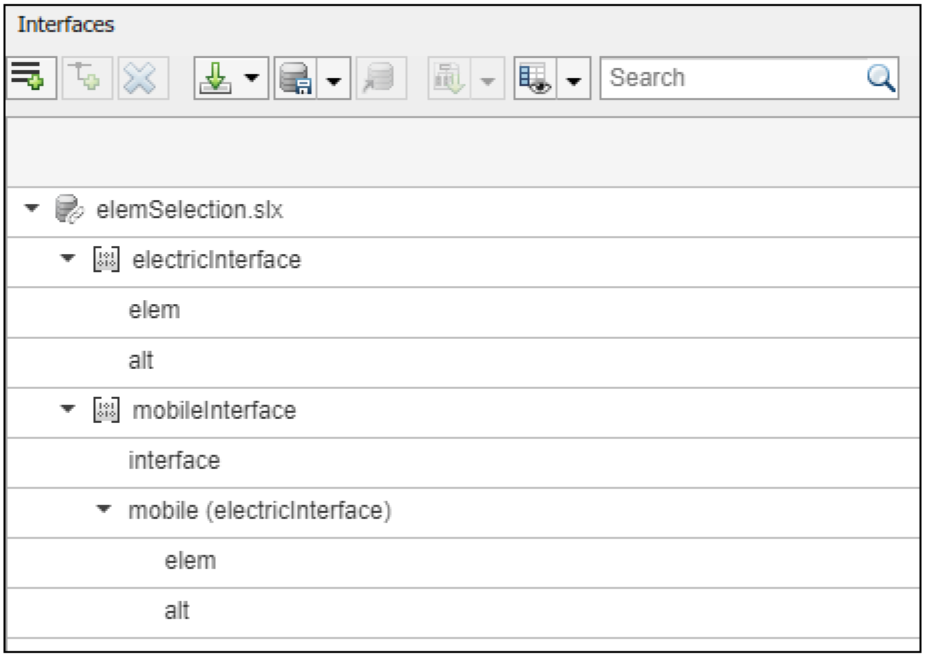
- DestinationType is how requirements are saved.
- DestinationID is where the requirement is located.
- Type is the requirement type.

Label	ID	SourceID	DestinationType	DestinationID	Type
rset#1	1	components:1	linktype_rmi_slreq	C:\Temp\rset.slreqx#1	Implement
rset#2	2	components:0	linktype_rmi_slreq	C:\Temp\rset.slreqx#2	Implement
rset#3	3	ports:1	linktype_rmi_slreq	C:\Temp\rset.slreqx#3	Implement
rset#4	4	ports:3	linktype_rmi_slreq	C:\Temp\rset.slreqx#4	Implement

A Simulink Requirements™ license is required to import requirement links into a System Composer architecture model.

Specify Elements on Architecture Port

In the connections table, you can specify different kinds of signal interface elements as source elements or destination elements. Connections can be formed from a root architecture port to a component port, from a component port to a root architecture port, or between two root architecture ports of the same architecture.



elemSelection.slx
electricInterface
elem
alt
mobileInterface
interface
mobile (electricInterface)
elem
alt

The nested interface element mobile.elem is the source element for the connection between an architecture port and a component port. The nested element mobile.alt is the destination element for the connection between an architecture port and a component port. The interface element mobile and the nested element mobile.alt are source elements for the connection between two architecture ports of the same architecture.

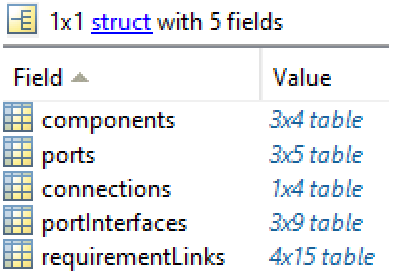
Name	ID	SourcePortID	DestPortID	SourceElement	DestinationElement
RootToComp1	1	5	4	mobile.elem	
RootToComp2	2	5	1	mobile.alt	
Comp1ToRoot	3	2	6		interface
Comp2ToRoot	4	3	6		mobile.alt
RootToRoot	5	5	6	mobile,mobile.alt	

Export an Architecture

To export a model, pass the model name and as an argument to the `exportModel` function. The function returns a structure containing four tables `components`, `ports`, `connections`, `portInterfaces`, and `requirementLinks`.

```
>> exportedSet = systemcomposer.exportModel(modelName)
```

You can export the set to MATLAB tables and then convert those tables to external file formats, including Microsoft® Excel® or databases.



1x1 struct with 5 fields

Field	Value
components	3x4 table
ports	3x5 table
connections	1x4 table
portInterfaces	3x9 table
requirementLinks	4x15 table

Update Reference Requirement Links from Imported File

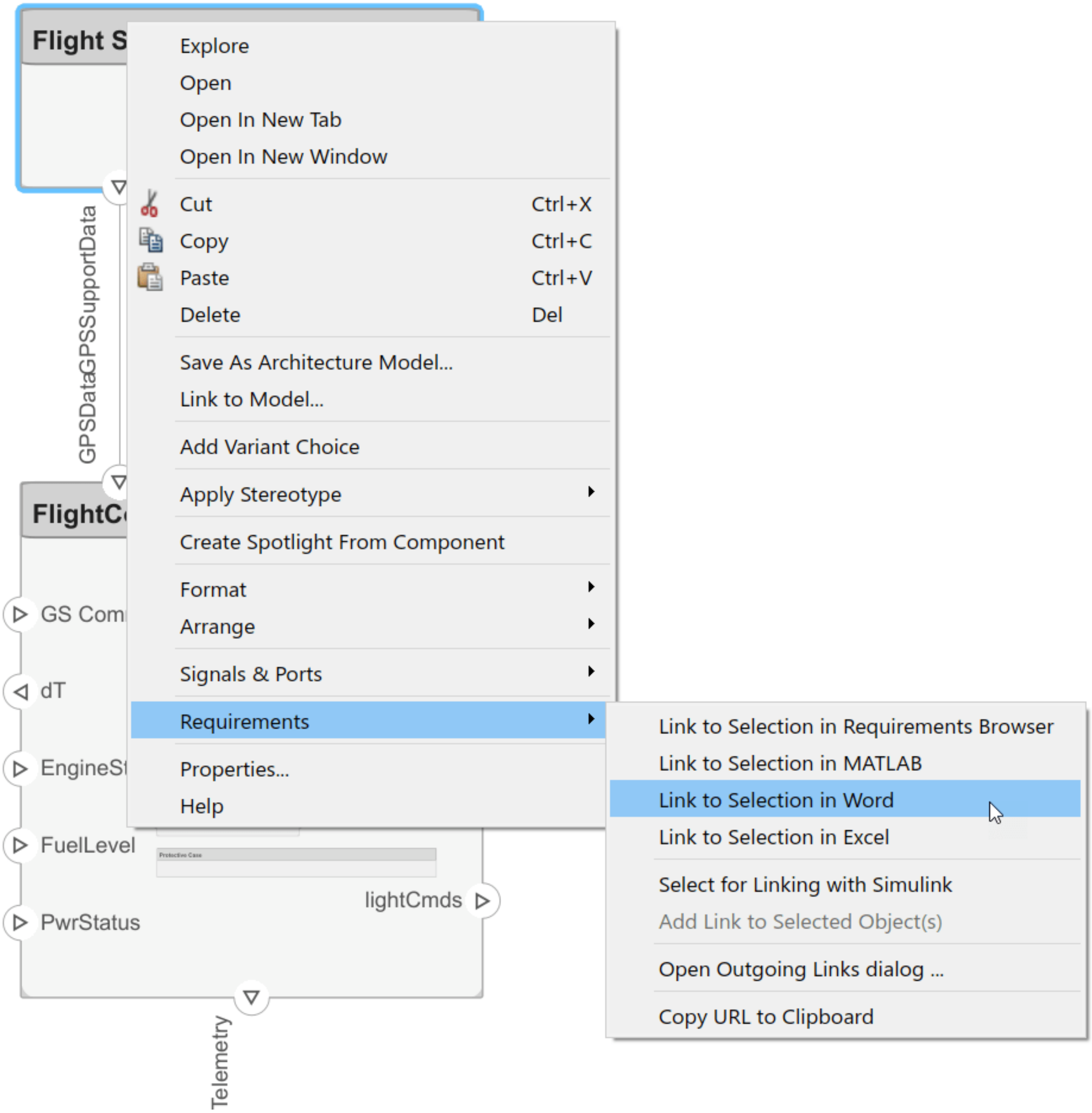
After importing requirement links from a file, update links to reference requirements for the model to make full use of the Simulink® Requirements™ functionality.

```
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
```

See Also

`exportModel` | `importModel` | `systemcomposer.io.ModelBuilder` | `updateLinksToReferenceRequirements`

More About

- “Import and Export Architectures” on page 6-39
- “Import System Composer Architecture Using Model Builder” on page 6-41

Display Component Hierarchy Using Hierarchy Views

This example shows how to use hierarchy views to visualize component hierarchy as a tree diagram with component stereotypes, stereotype properties, and the reference type a component instantiates.

Any component diagram view can be optionally represented as a hierarchy diagram. The hierarchy view displays the components in tree form. The hierarchy view shows the same set of components visible in the component diagram view, and the components displayed in the view are selected and filtered in the same way.

This example uses an architecture model representing a keyless entry system for a vehicle to show the hierarchy view. For more information about the keyless entry system, see “Modeling System Architecture of Keyless Entry System” on page 6-31.

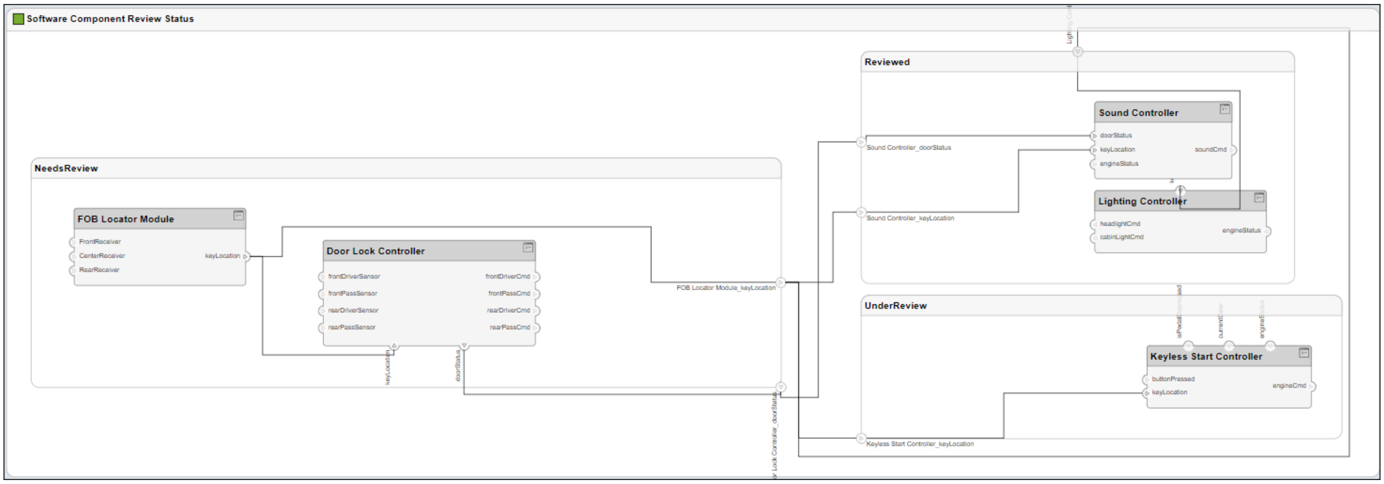
Switch Between Component Diagram and Hierarchy Diagram

1 To open the `scKeylessEntrySystem` project, use this command.

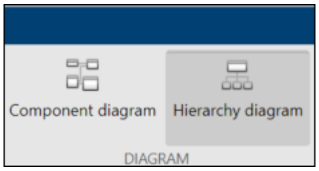
```
scKeylessEntrySystem
```

2 To open the architecture views, on the **Modeling** tab, select **Architecture Views**.

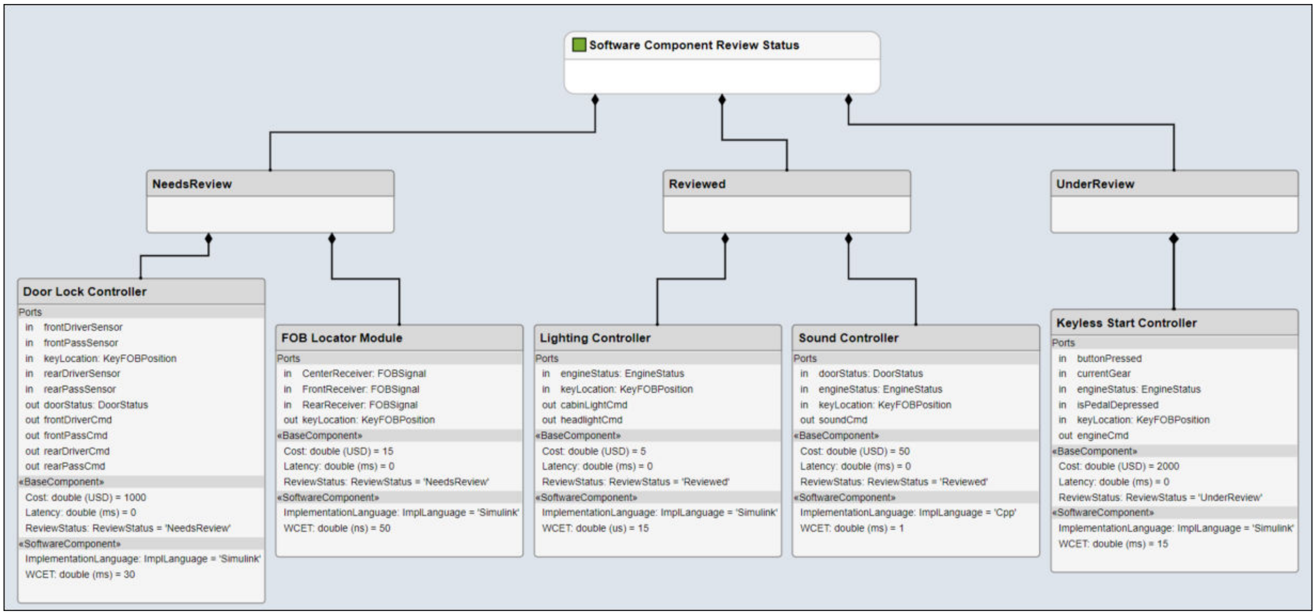
3 From the View Browser, select **Software Component Review Status** to display the component diagram.



4 On the **Views** tab, select **Hierarchy diagram**.



5 Observe the Hierarchy View that corresponds to the same set of components.


🟩 Software Component Review Status
NeedsReview
Reviewed
UnderReview
Door Lock Controller
Ports
in frontDriverSensor
in frontPassSensor
in keyLocation: KeyFOBPosition
in rearDriverSensor
in rearPassSensor
out doorStatus: DoorStatus
out frontDriverCmd
out frontPassCmd
out rearDriverCmd
out rearPassCmd
«BaseComponent»
Cost: double (USD) = 1000
Latency: double (ms) = 0
ReviewStatus: ReviewStatus = 'NeedsReview'
«SoftwareComponent»
ImplementationLanguage: ImplLanguage = 'Simulink'
WCET: double (ms) = 30
FOB Locator Module
Ports
in CenterReceiver: FOBSignal
in FrontReceiver: FOBSignal
in RearReceiver: FOBSignal
out keyLocation: KeyFOBPosition
«BaseComponent»
Cost: double (USD) = 15
Latency: double (ms) = 0
ReviewStatus: ReviewStatus = 'NeedsReview'
«SoftwareComponent»
ImplementationLanguage: ImplLanguage = 'Simulink'
WCET: double (ns) = 50
Lighting Controller
Ports
in engineStatus: EngineStatus
in keyLocation: KeyFOBPosition
out cabinLightCmd
out headlightCmd
«BaseComponent»
Cost: double (USD) = 5
Latency: double (ms) = 0
ReviewStatus: ReviewStatus = 'Reviewed'
«SoftwareComponent»
ImplementationLanguage: ImplLanguage = 'Simulink'
WCET: double (us) = 15
Sound Controller
Ports
in doorStatus: DoorStatus
in engineStatus: EngineStatus
in keyLocation: KeyFOBPosition
out soundCmd
«BaseComponent»
Cost: double (USD) = 50
Latency: double (ms) = 0
ReviewStatus: ReviewStatus = 'Reviewed'
«SoftwareComponent»
ImplementationLanguage: ImplLanguage = 'Cpp'
WCET: double (ms) = 1
Keyless Start Controller
Ports
in buttonPressed
in currentGear
in engineStatus: EngineStatus
in isPedalDepressed
in keyLocation: KeyFOBPosition
out engineCmd
«BaseComponent»
Cost: double (USD) = 2000
Latency: double (ms) = 0
ReviewStatus: ReviewStatus = 'UnderReview'
«SoftwareComponent»
ImplementationLanguage: ImplLanguage = 'Simulink'
WCET: double (ms) = 15

The single root of the hierarchy diagrams show a single root, which is the view specification itself. The root corresponds to the containing system box shown in the component diagram. The connections in the hierarchy diagram originate from the child components and end with a diamond symbol at the parent component.

See Also

More About

- “Create Architectural Views Programmatically” on page 1-45
- “Create Architecture Views Interactively” on page 1-37

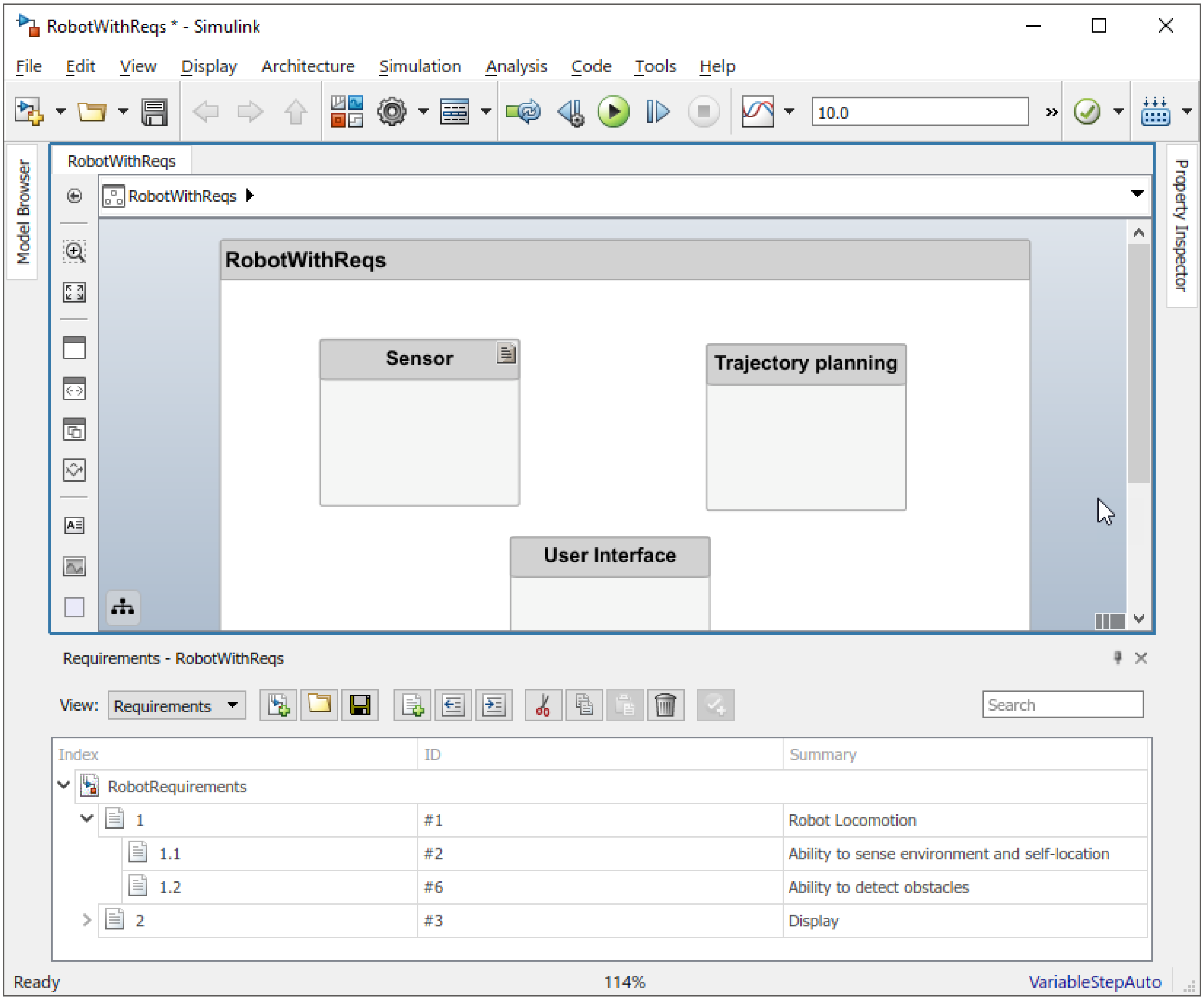
Requirements

Manage Requirements

Requirements are a collection of statements describing the desired behavior and characteristics of a system. Requirements ensure system design integrity and are achievable, verifiable, unambiguous, and consistent with each other. Each level of design should have appropriate requirements.

To enhance traceability of requirements, link system, functional, customer, performance, or design requirements to components and ports. Link requirements to each other to represent derived or allocated requirements. Manage requirements from the requirements perspective on an architecture model or through custom views. Assign test cases to requirements using the Test Manager for verification and validation. For more information on using Simulink Test™ with Simulink Requirements, see "Link to Test Cases from Requirements" (Simulink Requirements).

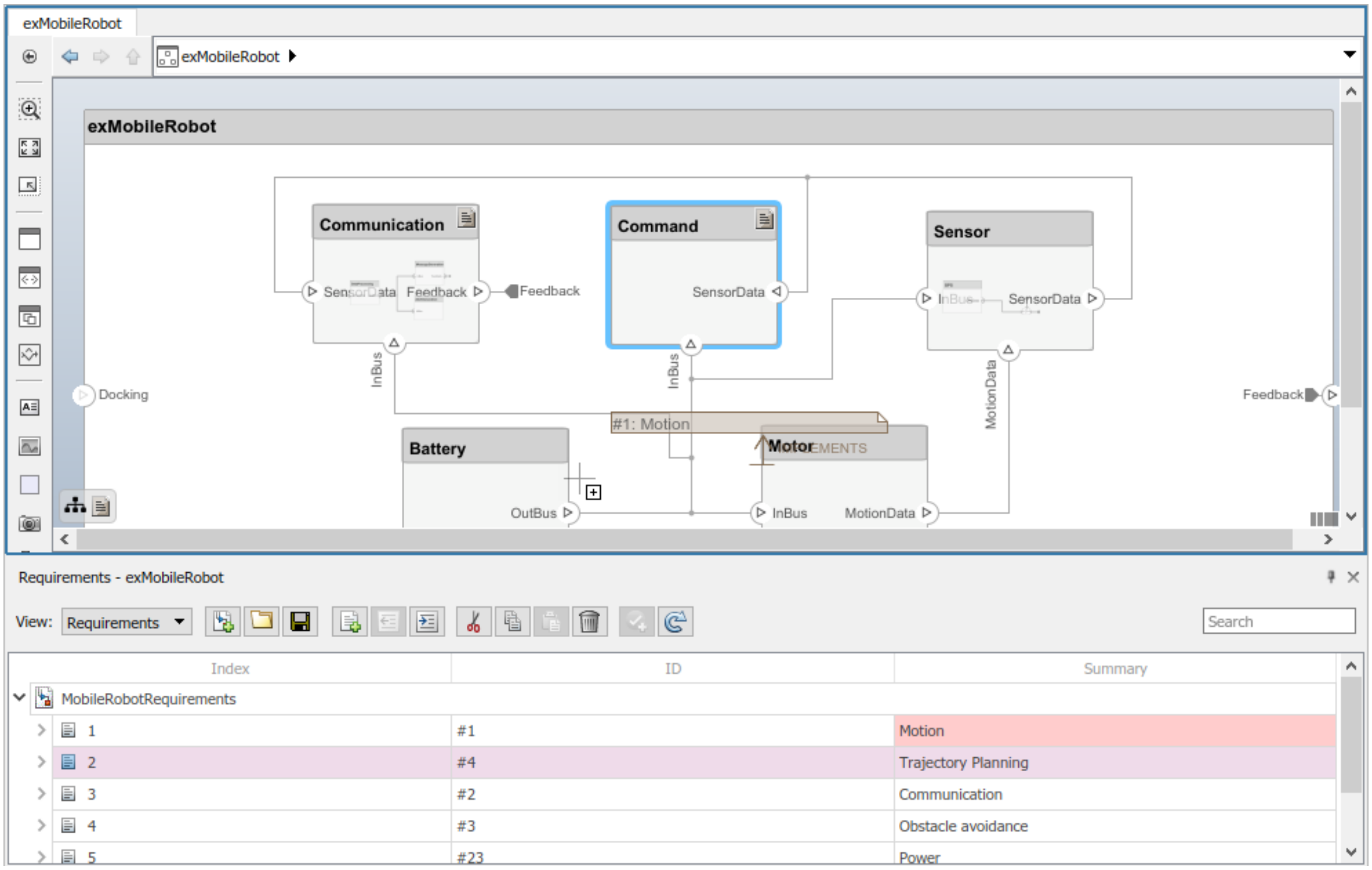
Manage requirements and architecture together in the **Requirements** perspective from Simulink Requirements. Select **Apps > Requirements Manager**.



Index	ID	Summary
RobotRequirements		
1	#1	Robot Locomotion
1.1	#2	Ability to sense environment and self-location
1.2	#6	Ability to detect obstacles
2	#3	Display

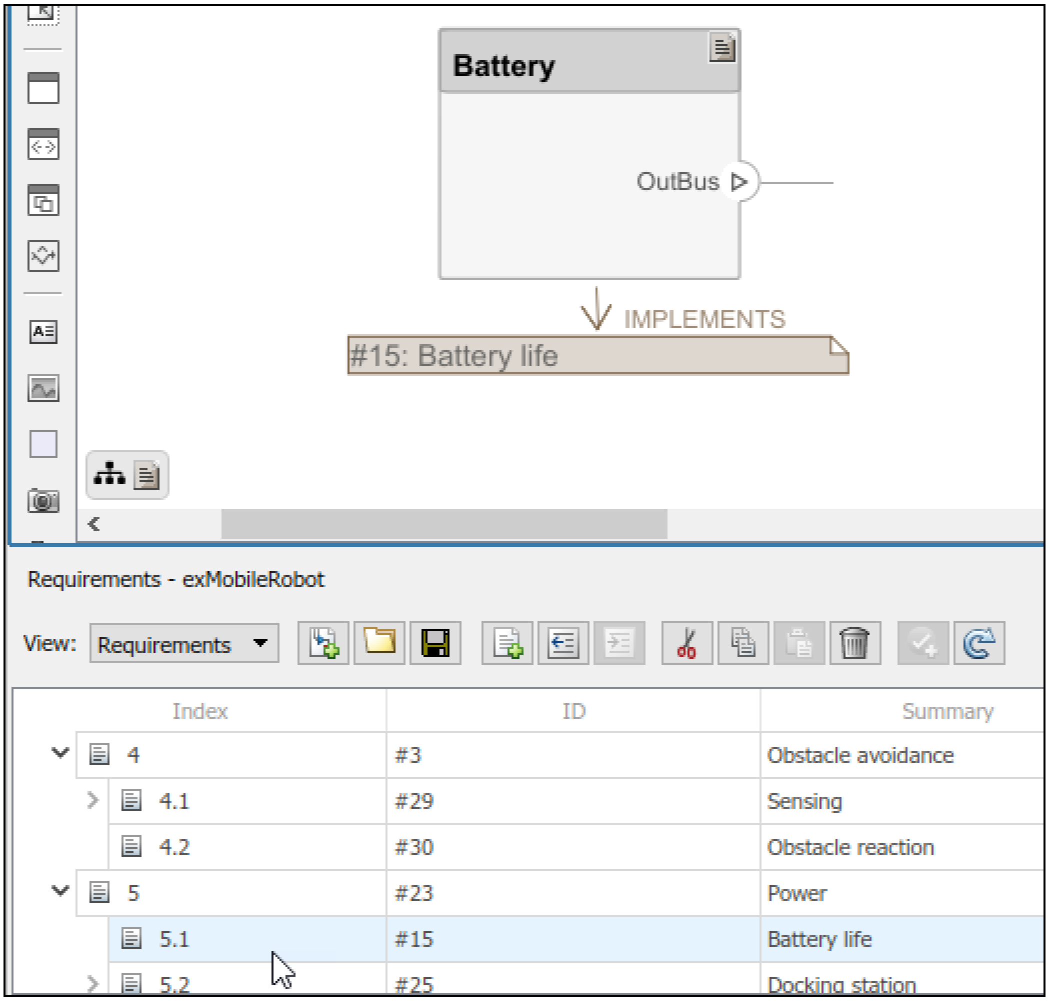
Ready 114% VariableStepAuto

When you click a component in the **Requirements** perspective, linked requirements are highlighted. Conversely, when you click a requirement, the linked components are shown.



Index	ID	Summary
MobileRobotRequirements		
1	#1	Motion
2	#4	Trajectory Planning
3	#2	Communication
4	#3	Obstacle avoidance
5	#23	Power

To directly create a link, drag a requirement onto a component or port.



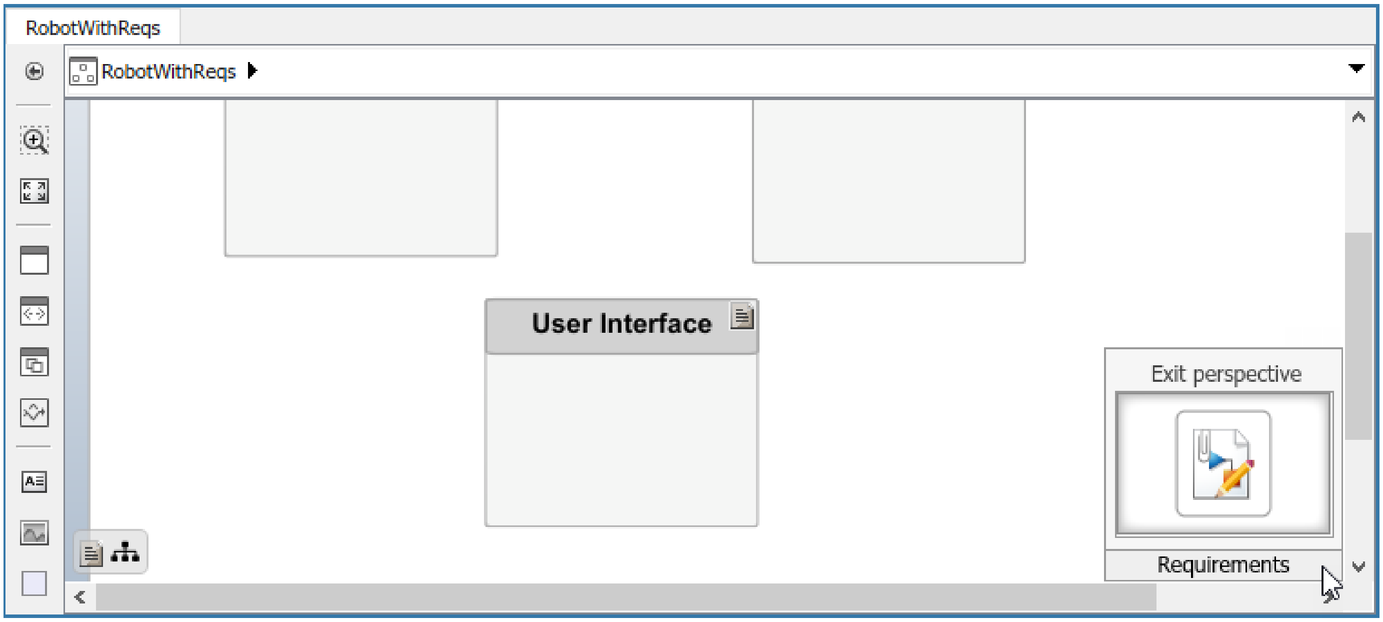
Requirements - exMobileRobot

View: Requirements

Index	ID	Summary
4	#3	Obstacle avoidance
4.1	#29	Sensing
4.2	#30	Obstacle reaction
5	#23	Power
5.1	#15	Battery life
5.2	#25	Docking station

You can close the annotation that shows the link as necessary. This action does not delete the link.

You can exit the **Requirements** perspective by clicking the perspectives menu on the lower-right corner of the architecture model and selecting **Exit perspective**.



For more information on managing requirements from external documents, see "Manage Navigation Backlinks in External Requirements Documents" (Simulink Requirements). To integrate the requirement links to the model, see "Update Reference Requirement Links from Imported File" on page 1-55.

See Also

`updateLinksToReferenceRequirements`

More About

- "Link and Trace Requirements" on page 6-25
- "Link Blocks and Requirements" (Simulink Requirements)
- "Import and Export Architectures" on page 6-39

Interface Management

Define Interfaces

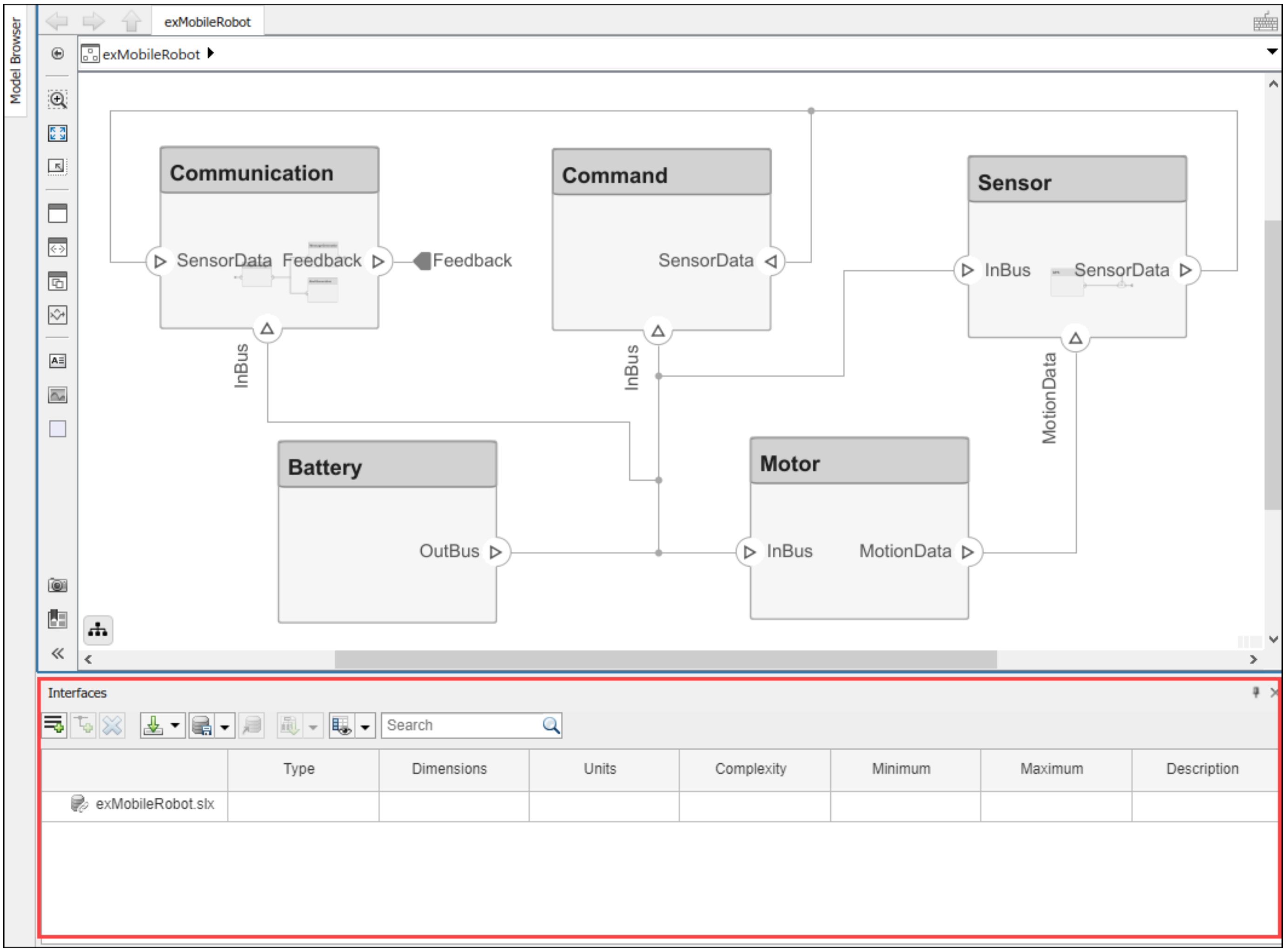
An interface defines the information that flows through a port. The same interface can be assigned to multiple ports. An interface can include elements that describe the properties of an interface signal. Interfaces represent the information that is shared through a connector and enters or exits a component 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.

An interface element is a piece of data that is transmitted across an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface. Examples of interface elements include:

- Pins or wires in a connector or harness.
- Messages transmitted across a bus.
- Data structures shared between components.

A system engineering solution includes a formal definition of the interfaces between components. A connection shows that two components have an output-to-input relationship, and an interface defines the type, dimensions, units, and structure of the data.

To show the Interface Editor, in the **Design** section, on the **Modeling** tab, select **Interface Editor**. The Interface Editor will open along the bottom pane.



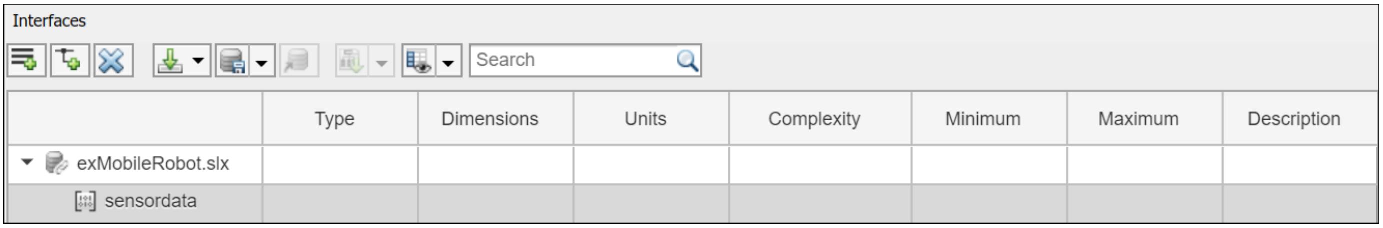
Interfaces

	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
exMobileRobot.slx							

Note The System Composer Interface Editor is a web-based widget and might appear blank when you first launch it. If this occurs, save the model and relaunch MATLAB with the command line option `-cefdisablegpu`.

Create Interface

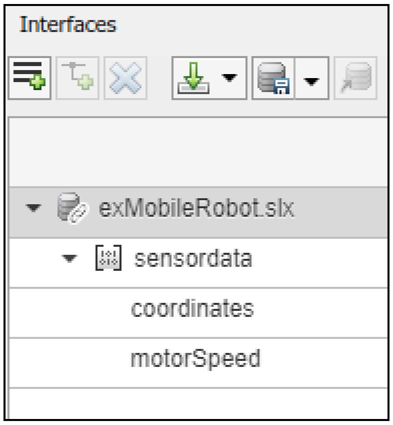
To add a new interface definition, click the icon. Name the interface.



Interfaces

	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
exMobileRobot.slx							
sensordata							

To add an element to the interface, click the icon. Interface and element names must be valid variable names.

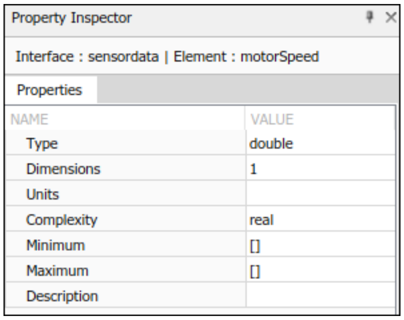


Interfaces

- exMobileRobot.slx
  - sensordata
    - coordinates
    - motorSpeed

You can delete interfaces and elements in the Interface Editor using the button.

You can view and edit the properties of an element in the Property Inspector. Right-click the interface element and select **Inspect Properties**. For interfaces, use the Property Inspector to apply stereotypes.



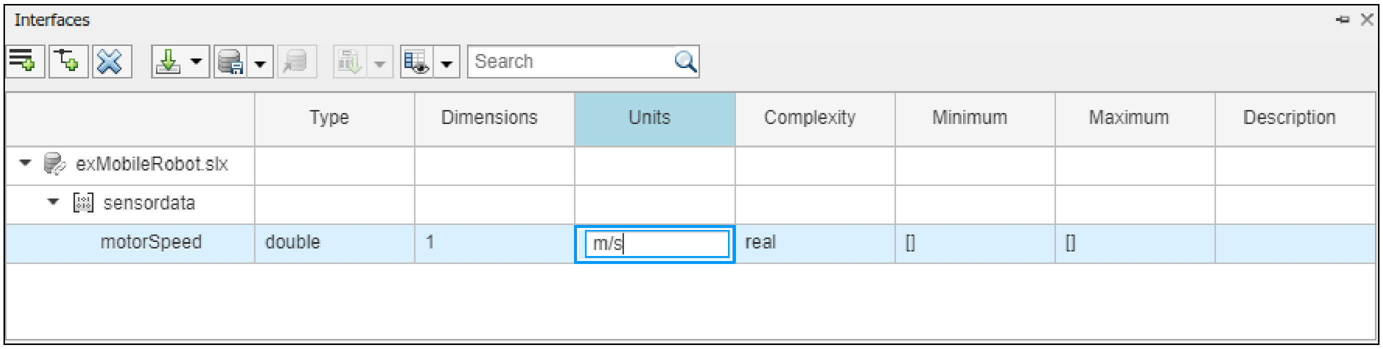
Property Inspector

Interface : sensordata | Element : motorSpeed

Properties

NAME	VALUE
Type	double
Dimensions	1
Units	
Complexity	real
Minimum	[]
Maximum	[]
Description	

For a comparative view, you can edit interface element properties from the Interface Editor columns.



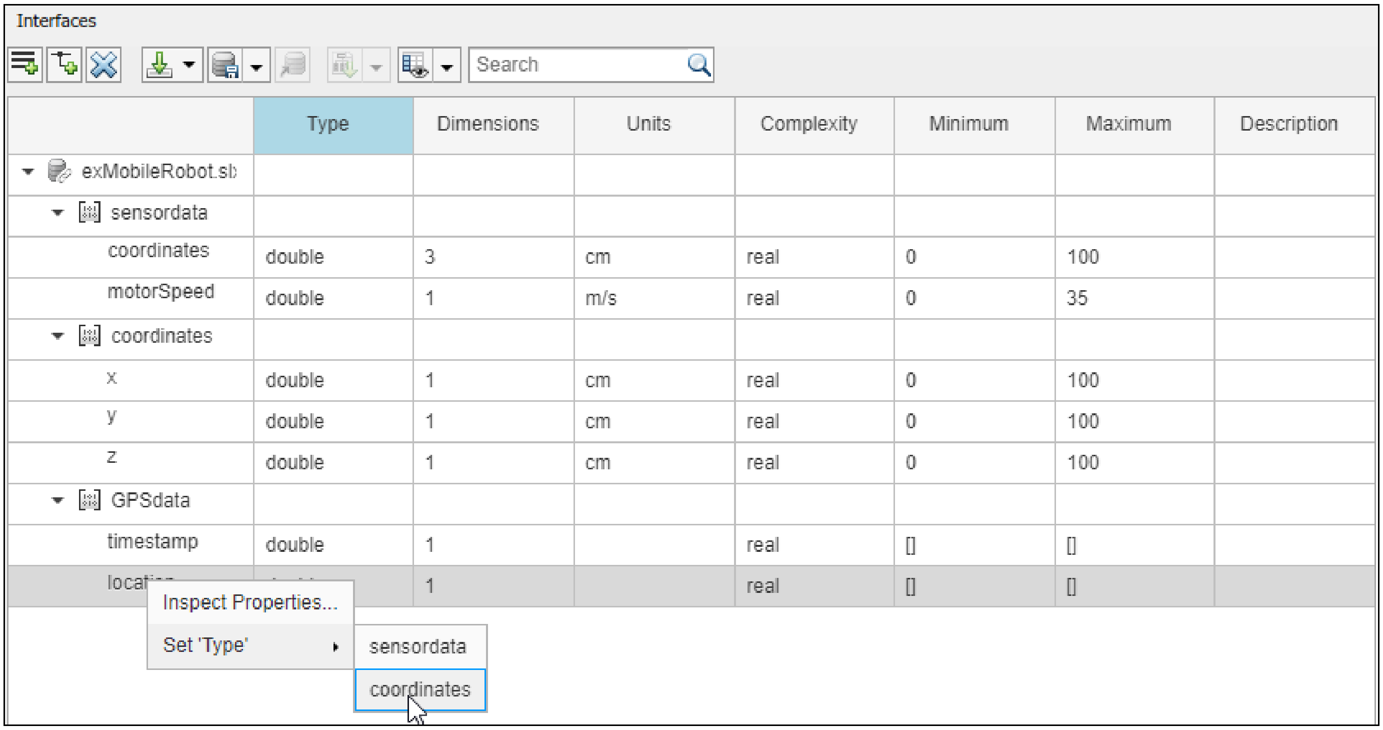
Interfaces

	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
exMobileRobot.slx							
sensordata							
motorSpeed	double	1	m/s	real	[]	[]	

Nested Interfaces

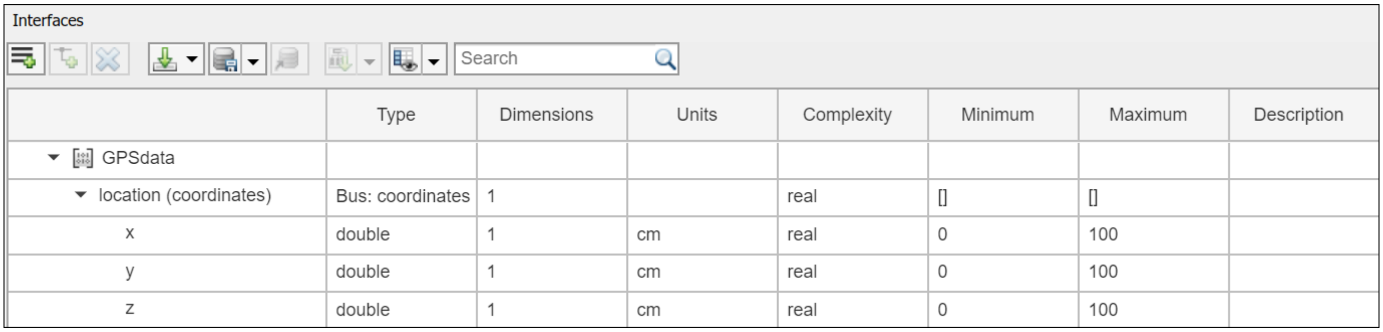
A nested interface contains another interface. Create a nested interface by assigning an interface as the type of an interface element. For information about the corresponding bus objects, see “Nest Bus Objects Using the Bus Editor”.

For example, let `coordinates` be an interface that consists of `x`, `y`, and `z` coordinates. `GPSdata` includes `location` and a `timestamp`. If the `location` element is in the same format as the `coordinates` interface, you can set its type to `coordinates`. Right-click `location` and select **Set 'Type' > coordinates**. The available interface options include all interfaces in the model, except the parent of the element.



	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
exMobileRobot.sl							
sensordata							
coordinates	double	3	cm	real	0	100	
motorSpeed	double	1	m/s	real	0	35	
coordinates							
x	double	1	cm	real	0	100	
y	double	1	cm	real	0	100	
z	double	1	cm	real	0	100	
GPSdata							
timestamp	double	1		real	[]	[]	
locat		1		real	[]	[]	

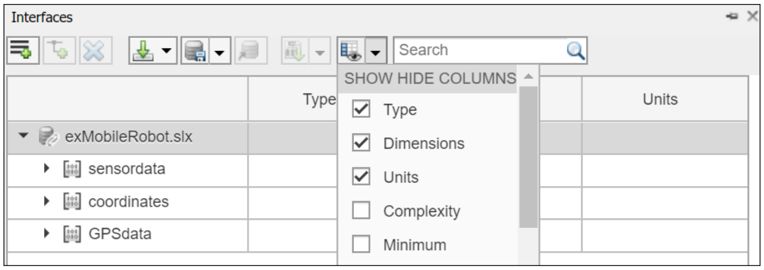
The nested interface displays the inherited interface elements.



	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
GPSdata							
location (coordinates)	Bus: coordinates	1		real	[]	[]	
x	double	1	cm	real	0	100	
y	double	1	cm	real	0	100	
z	double	1	cm	real	0	100	

Show and Hide Columns in the Interface Editor

To change the number of columns that display in the Interface Editor, select the icon. Select or deselect the desired columns to show or hide them.



SHOW HIDE COLUMNS

- ☑ Type
- ☑ Dimensions
- ☑ Units
- ☐ Complexity
- ☐ Minimum

See Also

`addElement` | `addInterface` | `createAnonymousInterface` | `getElement` | `getInterface` | `getInterfaceNames` | `removeElement` | `removeInterface`

More About

- “Assign Interfaces to Ports” on page 3-7
- “Save, Link, and Delete Interfaces” on page 3-12
- “Reference Data Dictionaries” on page 3-14

Assign Interfaces to Ports

A port interface describes the data that can be passed between ports. Interface elements within the interface describe characteristics of the data transmitted across the interface. Interface elements can describe the composition of an interface, messages transmitted, or data structures shared between components.

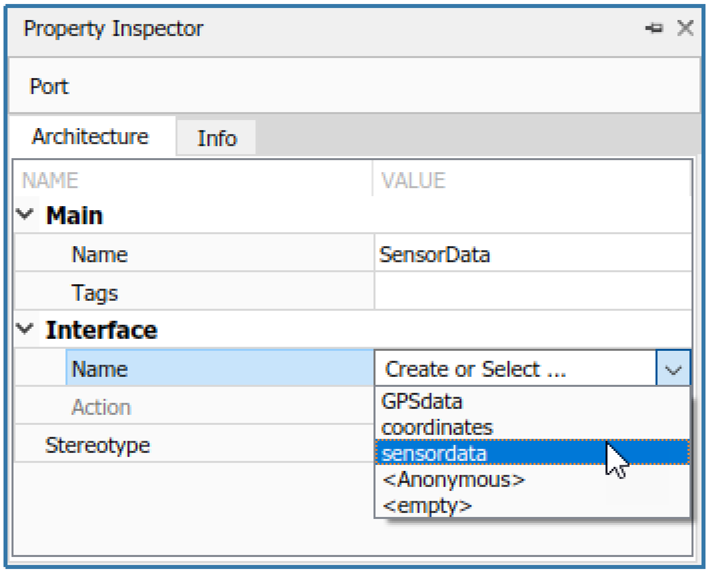
Use the Property Inspector to assign interfaces to one port at a time or the Interface Editor to assign interfaces to multiple ports.

You can connect components through ports and specify the source element or the destination element for the connection.

Incompatible interfaces on either end of a connection can be reconciled with an Adapter block using the "Interface Adapter" on page 3-19.

Associate a Port with an Interface in the Property Inspector

To open the Property Inspector, locate it in the toolstrip in the **Design** section drop down. To show the `SensorData` port properties, highlight the port in the model. Expand **Interface**, and select the `sensordata` interface in the **Name** drop-down menu.



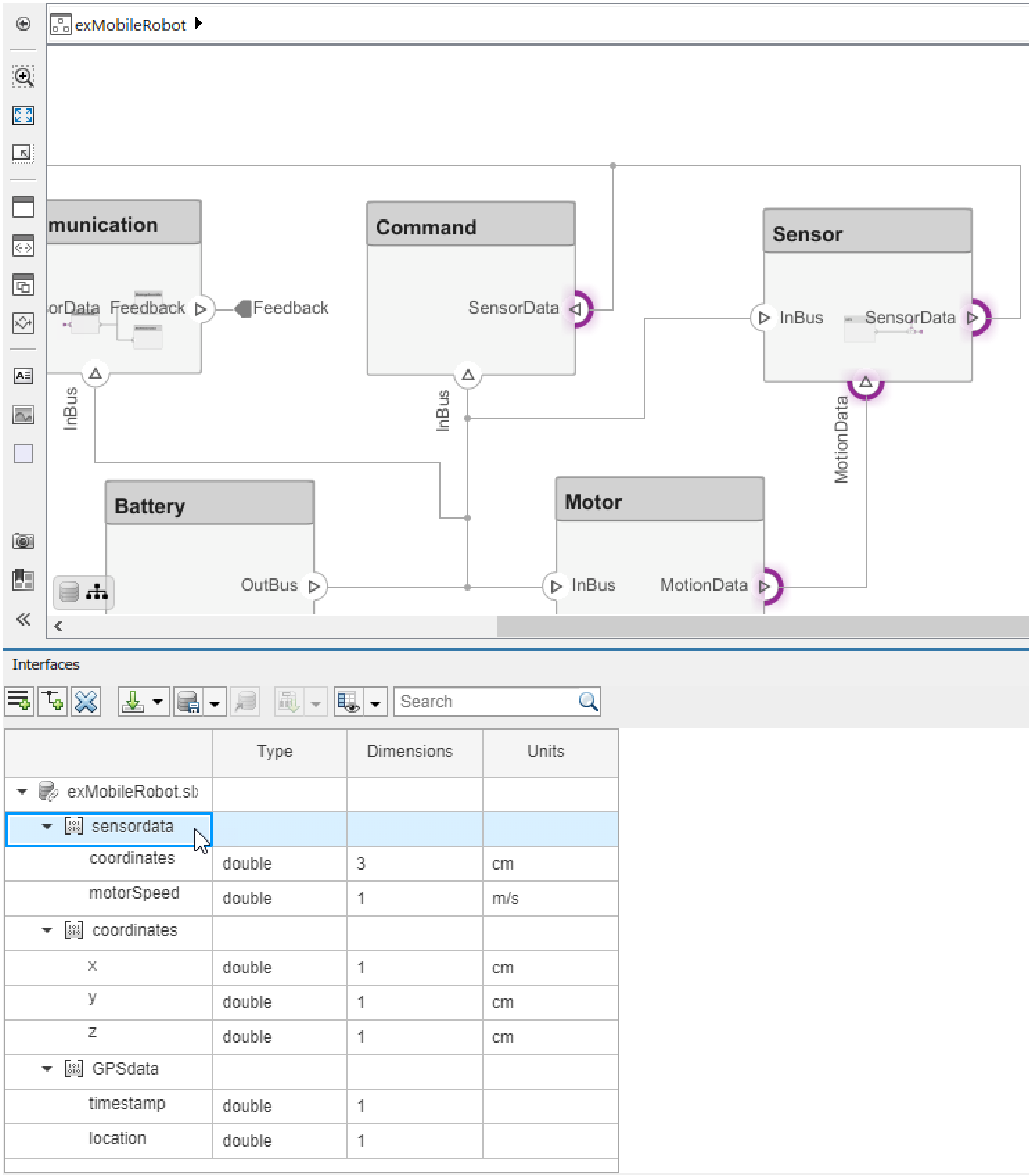
NAME	VALUE
Main	
Name	SensorData
Tags	
Interface	
Name	Create or Select ...
Action	
Stereotype	

You can select an interface in the model data dictionary (see "Define Interfaces" on page 3-2), or create an anonymous interface — an interface of unstructured data whose properties are valid for that port only. An anonymous interface does not have a structure, but does have prescribed properties such as **Type** and **Dimensions**. You can edit the properties of the anonymous interface in the Property Inspector.

Select Multiple Ports and Assign an Interface

Multiple ports, whether they are connected or not, can use the same interface definition. When you assign an interface to a port, it is automatically propagated to the connected ports, provided they do not already have assignments. To simplify batch assignments, select multiple ports, right-click the interface, and select `Assign to Selected Port(s)`.

Highlight the ports that use an interface definition by clicking the interface name in the Interface Editor.

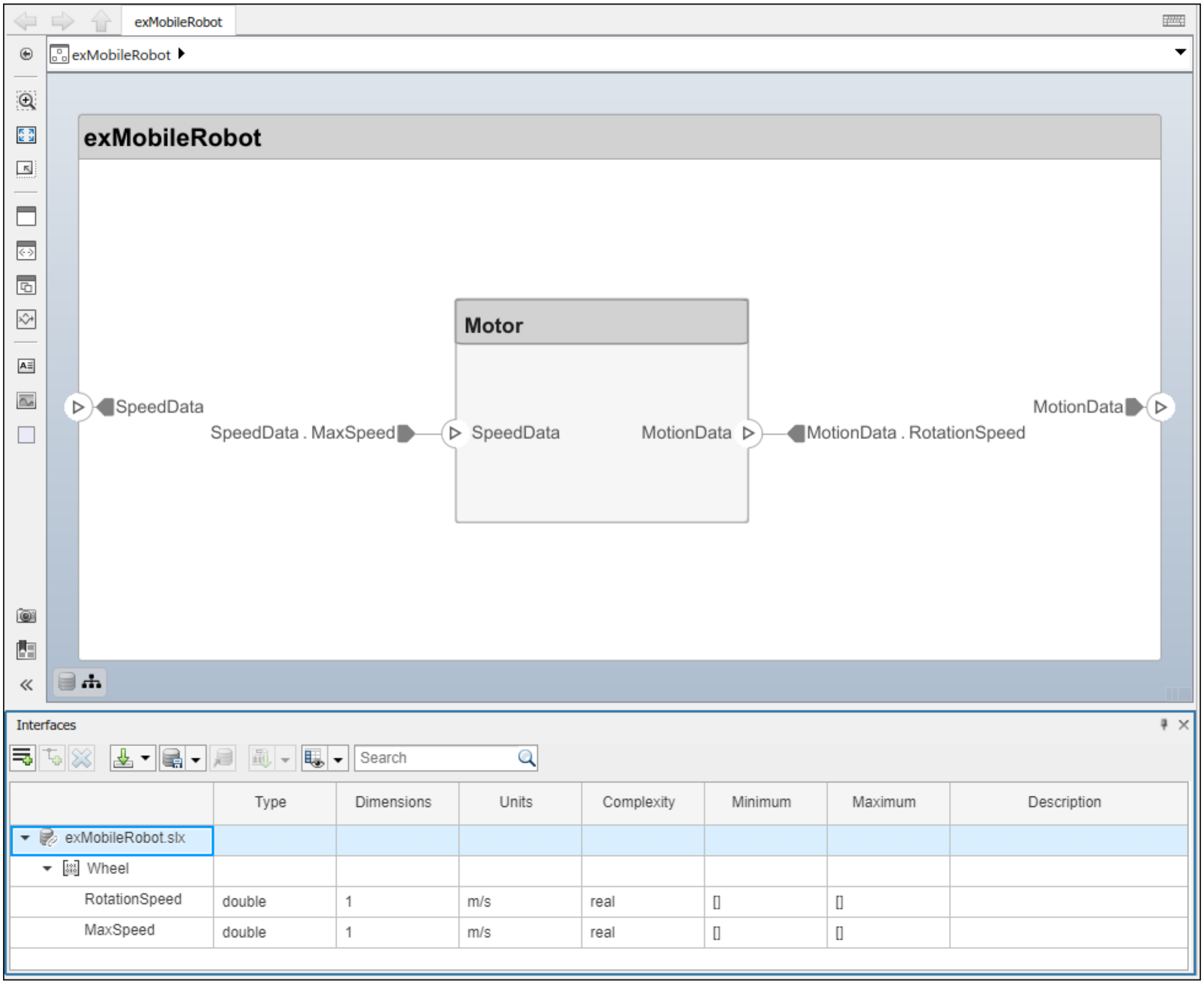


	Type	Dimensions	Units
exMobileRobot.sl			
sensordata			
coordinates	double	3	cm
motorSpeed	double	1	m/s
coordinates			
x	double	1	cm
y	double	1	cm
z	double	1	cm
GPSdata			
timestamp	double	1	
location	double	1	

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

For connections between the root architecture and a component within the architecture model, you can add a source element or destination element to the ports.

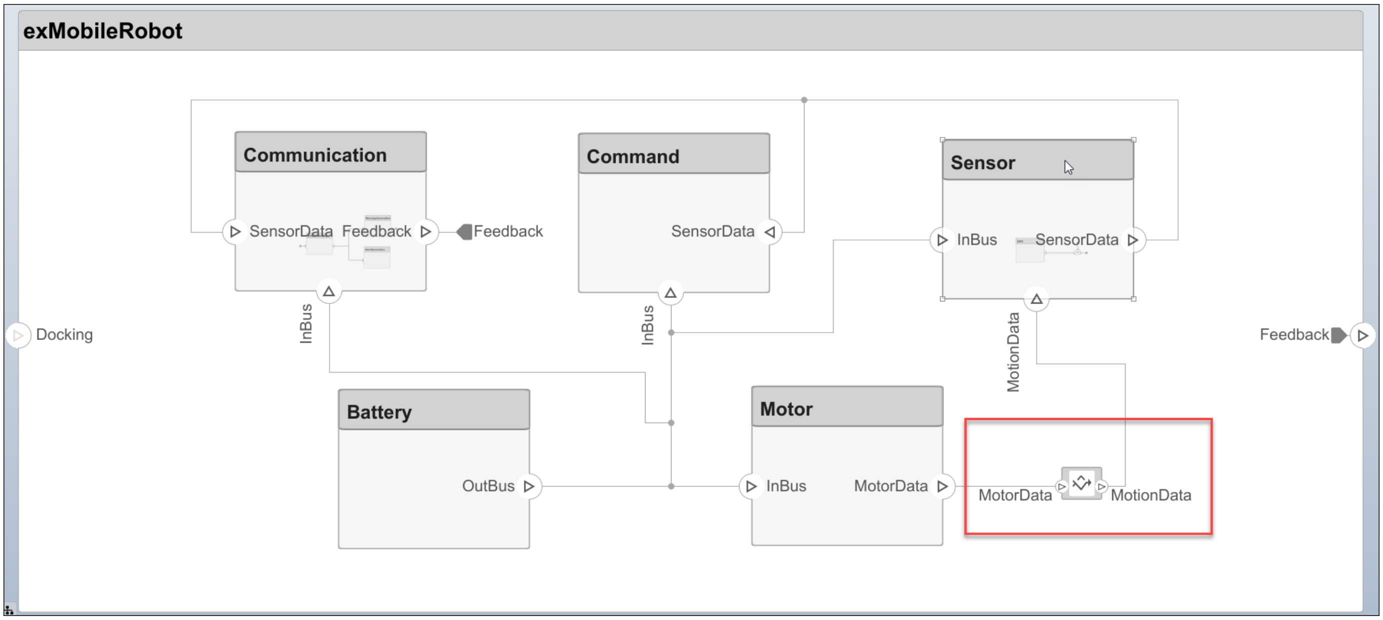
Create a component called `Motor` and connect it to the root architecture with ports named `MotionData` and `SpeedData`. Define the interface `Wheel` with the interface elements `RotationSpeed` and `MaxSpeed`. Assign the `Wheel` interface to the ports on the connection. Select the `MotionData` port name on the component and a dot and a list of signal interface elements will appear. Select the source element `RotationSpeed` from the list. Assign the `MaxSpeed` destination element to the `SpeedData` port.



	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
exMobileRobot.slx							
Wheel							
RotationSpeed	double	1	m/s	real	[]	[]	
MaxSpeed	double	1	m/s	real	[]	[]	

Reconcile Different Interfaces on Connected Ports using an Adapter block

A source port and the destination port to which it connects may be defined by different interfaces. Such a connection can represent an intermediate point in design, where components from different sources come together. To connect components with different interfaces, use an Adapter block from the component palette and the “Interface Adapter” on page 3-19.



Change the number of input ports on an Adapter block the same way you add and remove component ports. For more information, see “Ports” on page 1-9.

See Also

Functions
`connect` | `getDestinationElement` | `getSourceElement` | `setInterface`

Blocks
Adapter | Component

More About

- “Define Interfaces” on page 3-2
- “Save, Link, and Delete Interfaces” on page 3-12
- “Reference Data Dictionaries” on page 3-14
- “Interface Adapter” on page 3-19

Save, Link, and Delete Interfaces

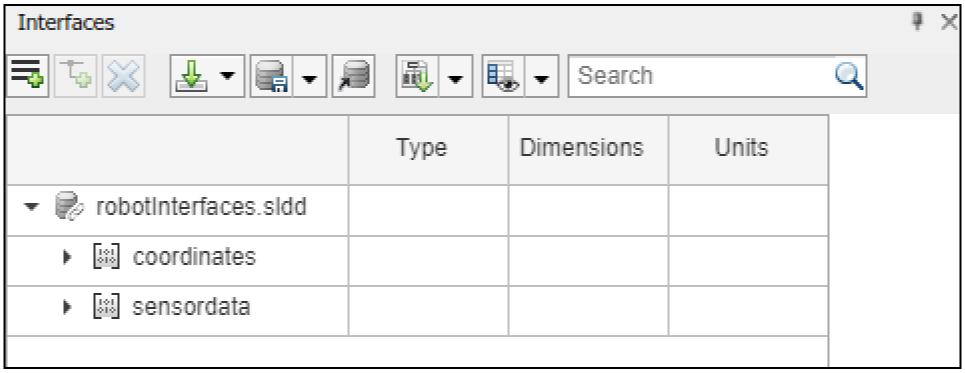
Engineering systems often share interface definitions across multiple components or subsystems.

Interfaces in System Composer can be stored either locally in a model or in a data dictionary, depending on the maturity of your system.

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.

By default, interfaces are stored within the architecture model and are not visible outside the model. If you are in the initial stages of building a system model, store interfaces locally to limit the number of files that need to be managed. However, if your model is mature to the point of leveraging componentization workflows like reference architectures and behaviors, storing interfaces in a data dictionary gives you the ability to share interface definitions across the model hierarchy.

Use the menu to save an interface to a new or existing data dictionary. To create a new data dictionary, select **Save to new dictionary**. Provide a dictionary name.

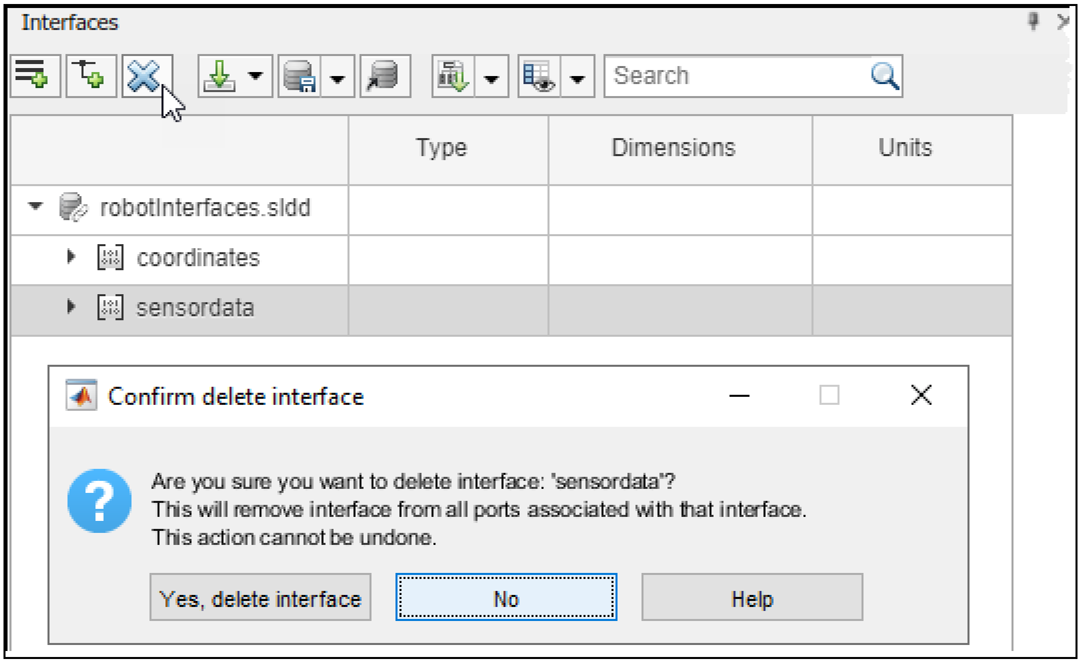


	Type	Dimensions	Units
robotInterfaces.sldd			
coordinates			
sensordata			

You can also add the interface definitions in the model to an existing data dictionary by selecting **Link existing dictionary**.

Use the button to import interface definitions from a Simulink bus object, either from a MAT-file or the workspace.

Delete an interface from a dictionary using the button. If the interface is already being used by ports in a currently open model, the software returns a warning message. The interface is then removed from any ports in the open model that are associated with the interface. Note that if an interface is deleted from a dictionary, upon opening another model that shares the dictionary, a warning will be presented on startup if the deleted interface is used by ports in that model. The Diagnostic Viewer offers an option to remove the deleted interface from all ports that are still using it. You can also select ports individually and delete their missing interfaces.



Are you sure you want to delete interface: 'sensordata'?
This will remove interface from all ports associated with that interface.
This action cannot be undone.

Note that a System Composer model and a data dictionary are separate artifacts. Even when the data dictionary is linked to the model, changes to the data dictionary (a `.sldd` file) must be saved separately from changes to the model (a `.slx` file). To save changes to a linked data dictionary, use the button and select `Save dictionary`. Once a data dictionary is saved, other models can use its interface definitions by linking to the data dictionary, allowing multiple models to share the same interface definitions.

See Also

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

More About

- “Define Interfaces” on page 3-2
- “Assign Interfaces to Ports” on page 3-7
- “Reference Data Dictionaries” on page 3-14

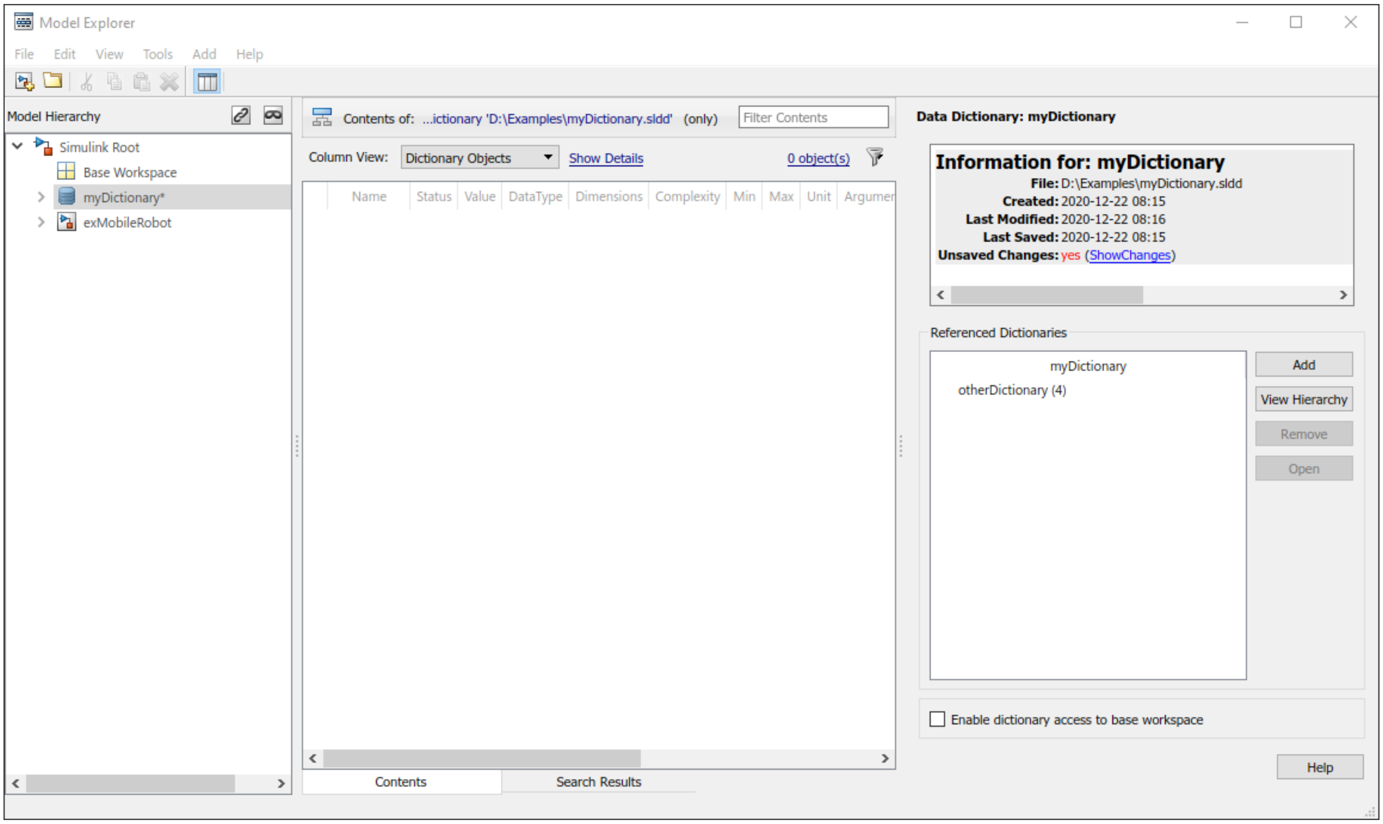
Reference Data Dictionaries

Add Referenced Data Dictionaries

Referenced dictionaries may be useful when multiple models need to share some, but not all, interface definitions. and to allow communication between the models. A data dictionary can reference one or more other data dictionaries. The interface definitions in the referenced dictionaries are visible in the parent dictionary and can be used by a model that is linked to the parent dictionary.

To add a dictionary reference, open the Model Explorer by clicking , or by selecting **Model Explorer** from the tab in the **Design** section of the **Modeling** tab.

On the right side of the Model Explorer window, click **Add**, then select the file name of the data dictionary to add as a referenced dictionary. To remove a dictionary reference, highlight the referenced dictionary, then click **Remove**.



Contents of: ...ictionary 'D:\Examples\myDictionary.sldd' (only)

Column View: Dictionary Objects

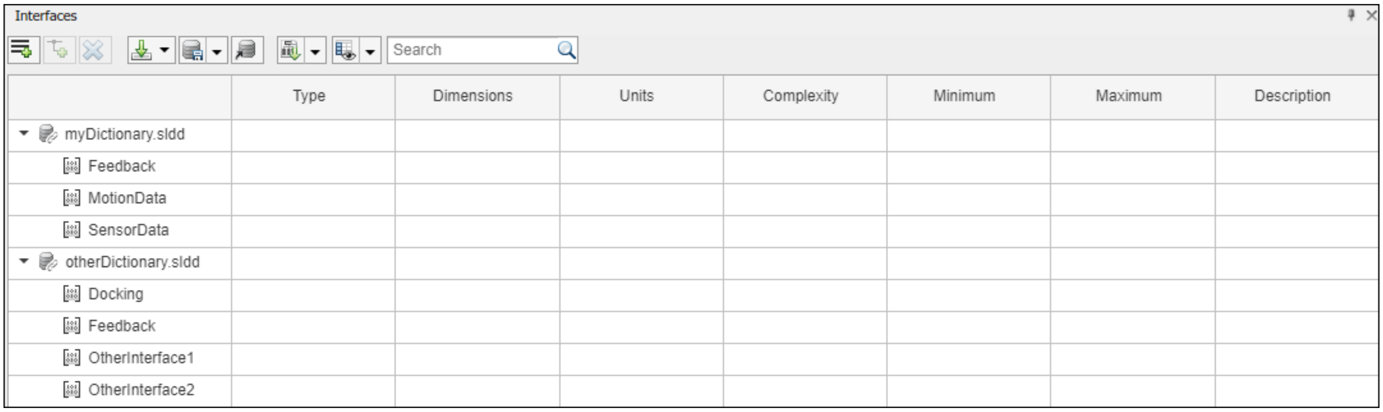
Data Dictionary: myDictionary

Information for: myDictionary
File: D:\Examples\myDictionary.sldd
Created: 2020-12-22 08:15
Last Modified: 2020-12-22 08:16
Last Saved: 2020-12-22 08:15
Unsaved Changes: yes (ShowChanges)

myDictionary
otherDictionary (4)

□ Enable dictionary access to base workspace

The Interface Editor shows all interfaces accessible to a model, grouped based on their data dictionary files. In this example, `myDictionary.sldd` is the data dictionary linked to the model, and `otherDictionary.sldd` is a referenced dictionary.



	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
myDictionary.sldd							
Feedback							
MotionData							
SensorData							
otherDictionary.sldd							
Docking							
Feedback							
OtherInterface1							
OtherInterface2							

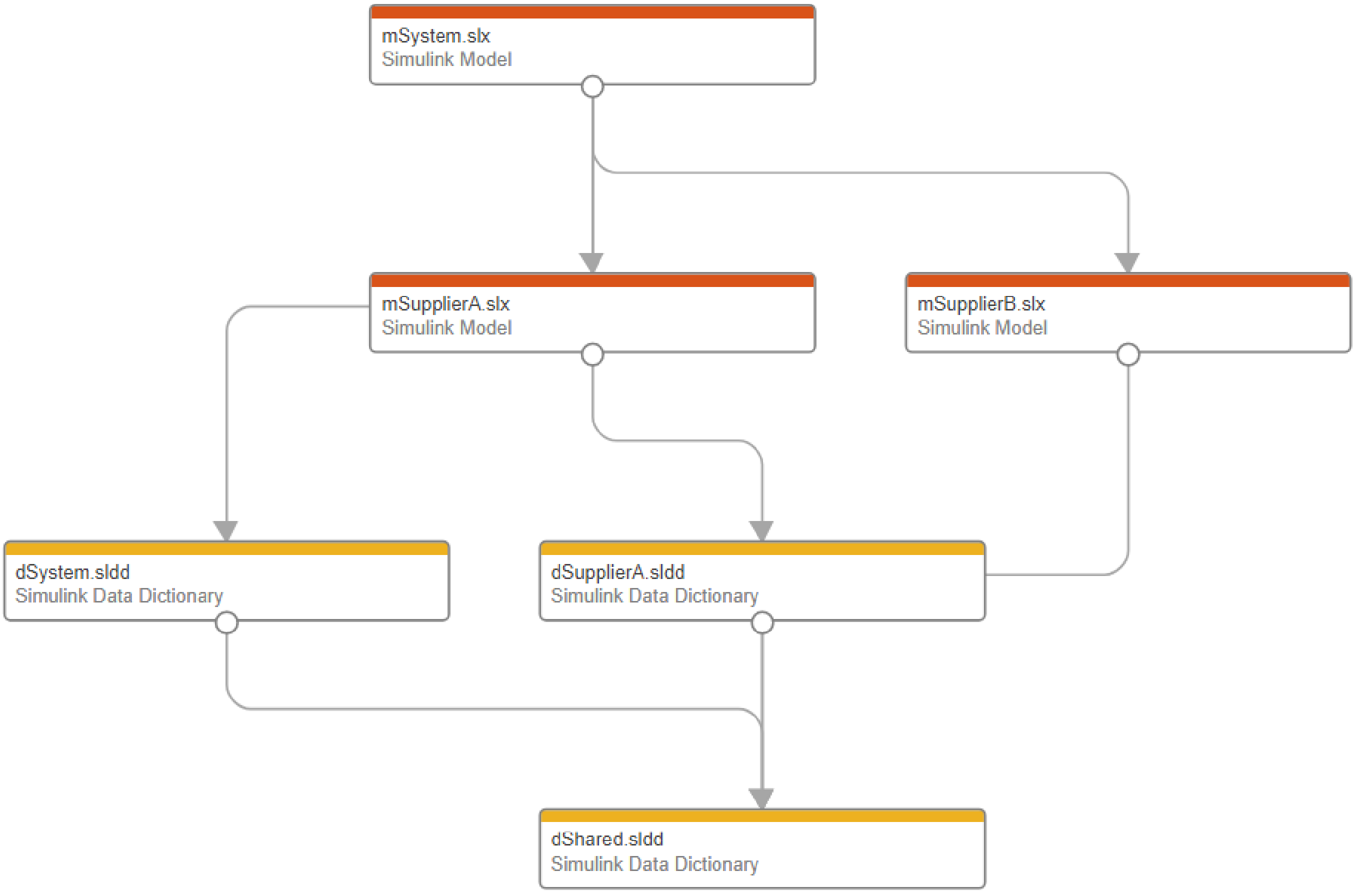
The model can use any of the interfaces listed. However, you cannot modify the contents of the referenced dictionaries from the model.

Note that referenced dictionaries can reference other data dictionaries. A model that links to a dictionary has access to all interface definitions in referenced dictionaries, including indirectly referenced dictionaries.

Use Referenced Data Dictionaries for Projects with Multiple Models

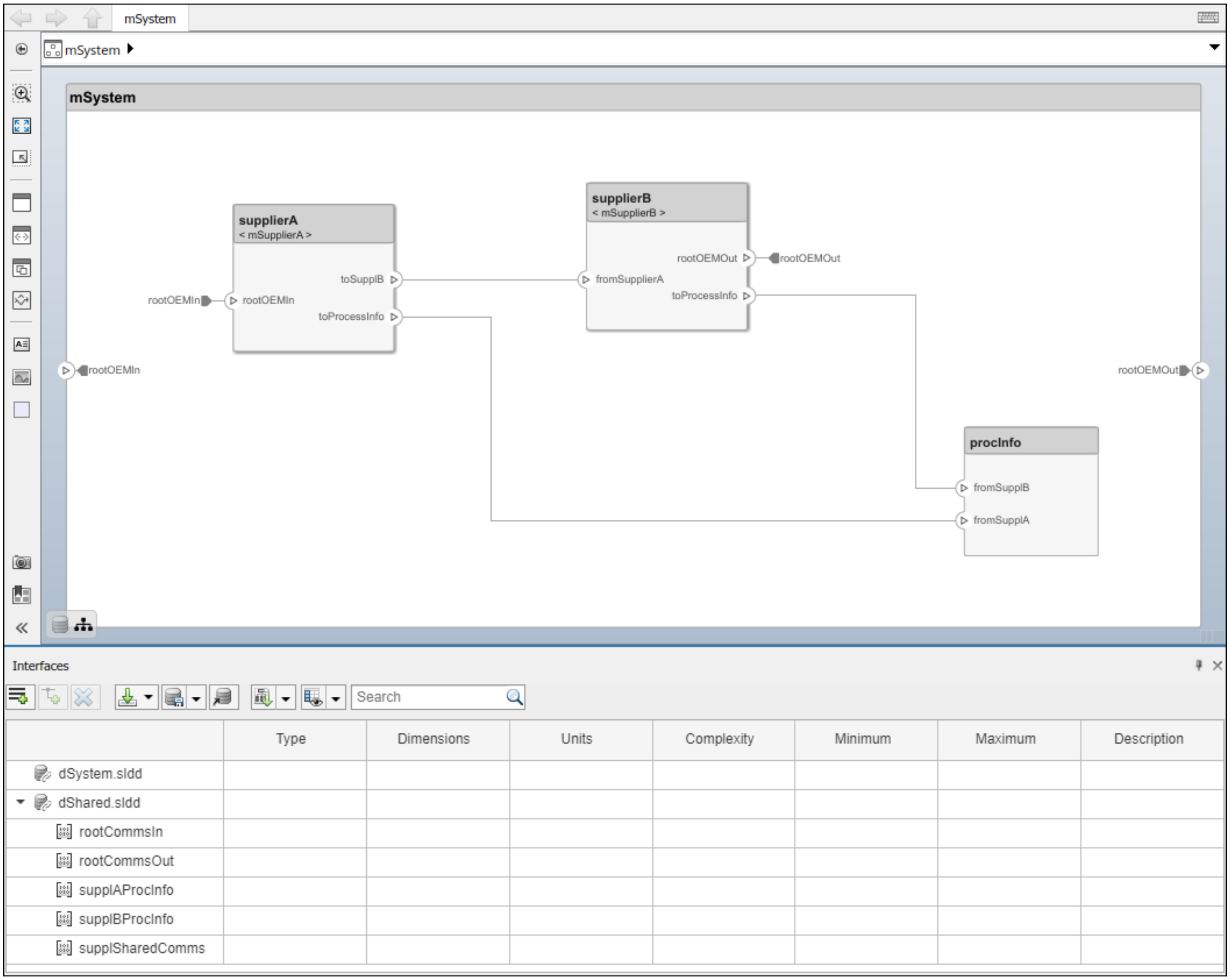
A project may contain multiple models, and it may be useful for the models to share interface definitions that are relevant to data flows and other communications between models. At the same time, each model may have interface definitions that are relevant only to its internal operations. For example, different components of a system may be represented by different models, with different teams or different suppliers working on each model, with a system integrator working on the "top" model that incorporates the various components. Referenced data dictionaries provide a way for models to share some but not all interface definitions.

In such a multiple-team project, set up a "shared artifacts" data dictionary to store interface definitions that will be shared by different teams, then set up a data dictionary for each model within the project to store its own interface definitions. Each data dictionary can then add the shared data dictionary as a referenced data dictionary. Alternatively, if a model does not need its own interface definitions, that model can link directly to the shared data dictionary.



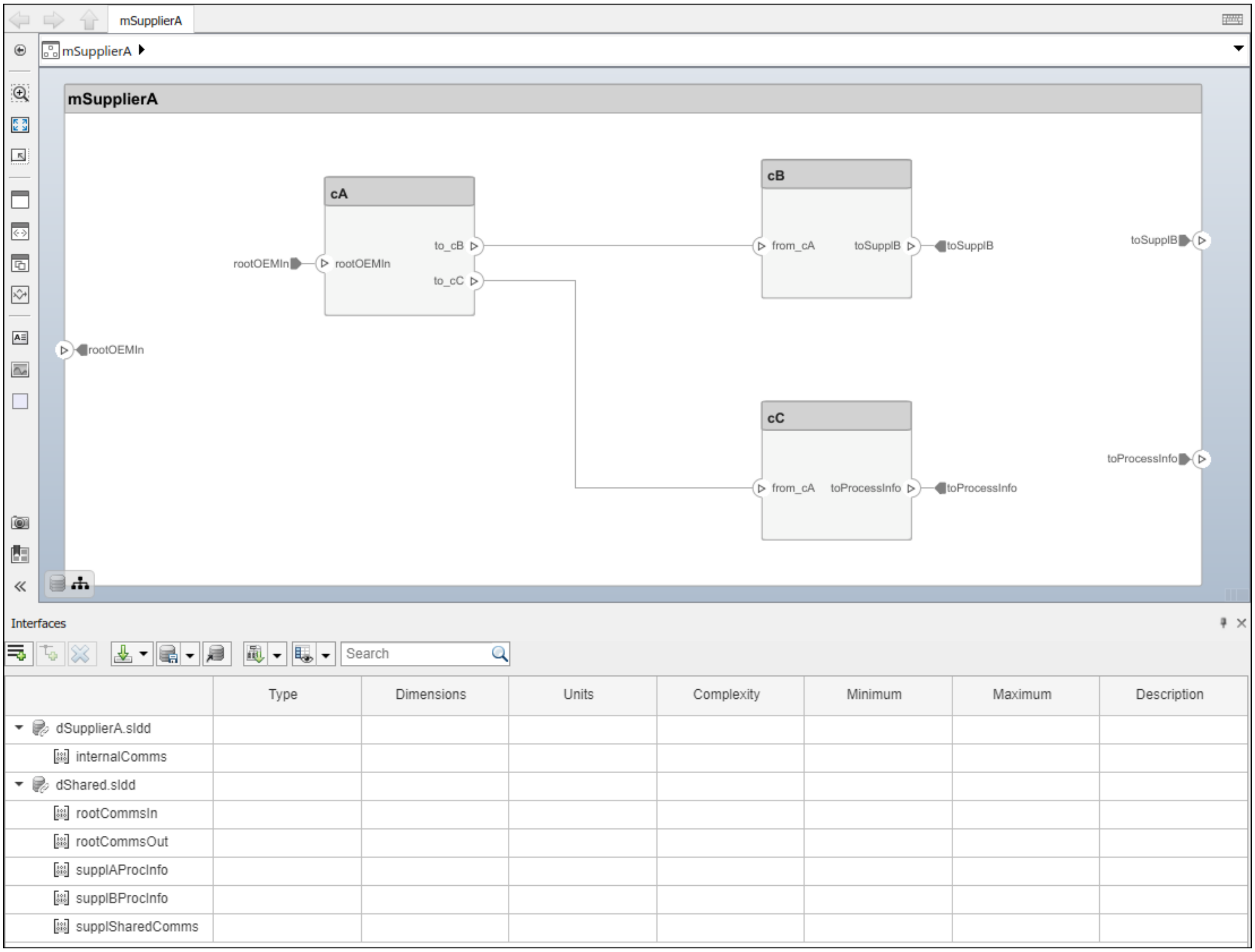
The above diagram depicts a project with three models. The model `mSystem.slx` represents a system integration model, and `mSupplierA.slx` and `mSuppierB.slx` represent supplier models. The data dictionary `dShared.sldd` contains interface definitions shared by all the models. The system integration model is linked to the data dictionary `dSystem.sldd`, and the Supplier A model is linked to the data dictionary `dSupplierA.sldd`; each data dictionary contains interface definitions relevant to the corresponding model's internal workflow. The data dictionaries `dSystem.sldd` and `dSupplierA.sldd` both reference the shared dictionary `dShared.sldd`. The Supplier B model, by contrast, is linked directly to the shared dictionary `dShared.sldd`. In this way, all three models have access to the interface definitions in `dShared.sldd`.

The following diagrams show the system integration model `mSystem`, along with the Interface Editor. Interface definitions contained in the referenced dictionary `dShared` are associated with the ports used to communicate between the models `mSupplierA` and `mSupplierB` and the rest of the system integration model.



	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
dSystem.sldd							
dShared.sldd							
rootCommsIn							
rootCommsOut							
supplAProcInfo							
supplBProcInfo							
supplSharedComms							

The following diagrams show the supplier model `mSupplierA`, along with the Interface Editor. Interface definitions contained in the referenced dictionary `dShared` are associated with the ports used to communicate externally, while interface definitions in the private dictionary `dSupplierA` are associated with ports whose use is internal to the `mSupplierA` model.



	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
dSupplierA.sldd							
internalComms							
dShared.sldd							
rootCommsIn							
rootCommsOut							
supplAProcInfo							
supplBProcInfo							
supplSharedComms							

See Also

`addReference` | `removeReference`

More About

- “Define Interfaces” on page 3-2
- “Assign Interfaces to Ports” on page 3-7
- “Save, Link, and Delete Interfaces” on page 3-12

Interface Adapter

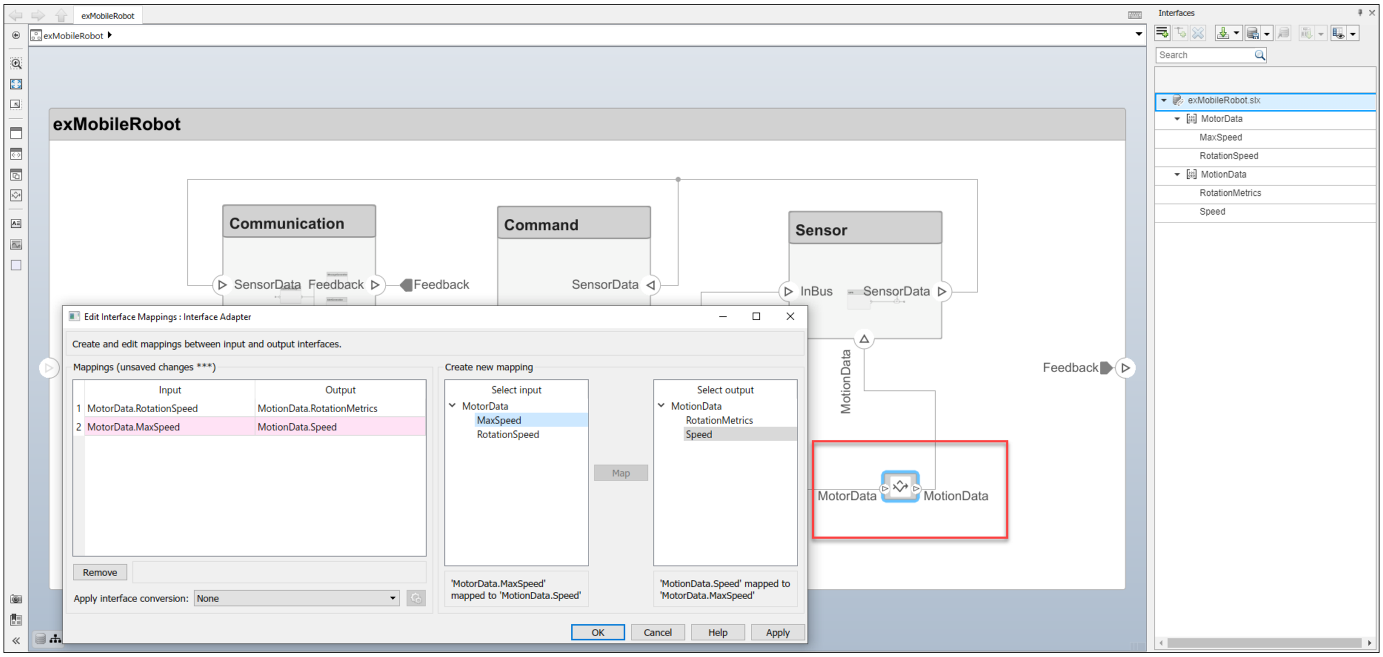
An Adapter block helps connect two components with incompatible port interfaces by mapping between the two interfaces. Launch the **Interface Adapter** by double-clicking an Adapter block on the connection between the ports.

Use the Interface Adapter to map interface elements between two ports. You can also use the Interface Adapter to apply an interface conversion to use unit delays to break algebraic loops, or to insert a rate transition for different sample time rates.

Map Similar Interfaces

When two connected components with Simulink behaviors have the same number of signals with different names, use an Adapter block and the Interface Adapter to define the port connections.

1. Add an Adapter block to your model on the connection between the two components.
2. Double-click the block to open the Interface Adapter dialog box.
3. In the **Select input** box, select an interface element. In the **Select output** box, select an interface element.
4. Click the **Map** button.



Edit Interface Mappings : Interface Adapter

Create and edit mappings between input and output interfaces.

Mappings (unsaved changes ***)

	Input	Output
1	MotorData.RotationSpeed	MotionData.RotationMetrics
2	MotorData.MaxSpeed	MotionData.Speed

Create new mapping

Select input: MotorData > MaxSpeed, RotationSpeed

Select output: MotionData > RotationMetrics, Speed

'MotorData.MaxSpeed' mapped to 'MotionData.Speed'

'MotionData.Speed' mapped to 'MotorData.MaxSpeed'

Apply interface conversion: None

Use Unit Delay to Break Algebraic Loop

When connecting two components with port connections in both directions, an algebraic loop can occur. To break the algebraic loop, use an Adapter block to insert a unit delay between the components.

1. Add an Adapter block to your model on the connection between the two components.

2. Double-click the block to open the Interface Adapter dialog box.
3. From the **Apply interface conversion** list, select `UnitDelay`.

Use Rate Transition Between Simulink Behaviors

When connecting two Reference Components, the Simulink models they reference can have different sample time rates. For compatibility, use an Adapter block to insert a rate transition between the components.

1. Add an Adapter block to your model on the connection between the two components.
2. Double-click the block to open the Interface Adapter dialog box.
3. From the **Apply interface conversion** list, select `RateTransition`.

See Also

Blocks
Adapter

More About

- “Define Interfaces” on page 3-2
- “Save Simulink.Bus Objects”
- “Assign Interfaces to Ports” on page 3-7

Define Architectural Properties

Define Profiles and Stereotypes

To verify structural and functional requirements, you must capture nonfunctional properties on elements in an architecture model. To capture these properties, use stereotyping.

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.

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.

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.

For example, if there is a limit on the total power consumption of a system, the model must be able to capture the power rating of each electrical component. To define component-specific property values requires extending built-in model element types with properties corresponding to requirements. In this case, an electrical component type as an extension of components is a stereotype. By extending the definition of regular components, you introduce a custom modeling language and framework that includes specific concepts and terminologies important for the architecture model. Capturing the individual properties also sets the scene for early parametric analyses and to define custom views.

You can define default stereotypes in a profile to be added to any new element in a model with that applied profile. Stereotype-based styling enhances the appearance of the model based upon specific features each element represents.

System Composer provides these architectural model elements to describe an architecture model:

- Component
- Port
- Connector
- Interface

You can view and edit the properties of each element in the architecture model using the Property Inspector. Open the Property Inspector using **View > Property Inspector**.

You can author profiles using the Profile Editor. Profiles are saved separately from the architecture model as `.xml` files and are available to all architecture models.

When you create a profile, you define:

- Stereotypes — Customize built-in model element types.
- Property sets — Add analysis properties to an architecture model element.
- Data types, units, dimensions, etc. — Define property values.

You can define stereotypes to extend built-in elements and capture additional data about an element. Element stereotypes define the class of the elements to which they apply. For example, a

MechanicalComponent stereotype with properties such as Weight and Volume applies only to components, and not to ports, connectors, or interfaces.

A stereotype does not have to define a class. For example, a ProjectItem stereotype can add generic properties such as CatalogNumber or UnitCost, a BorrowedItem stereotype can add properties such as BorrowedSource and ReturnDeadline. A model element can have multiple stereotypes.

Stereotypes can extend other stereotypes to include their properties through an inherited mechanism. For example, a UserInterface stereotype can be an extension of a SoftwareComponent stereotype, and add a property called ScreenResolution.

You can collect these stereotypes in profiles to import into the model.

Create a Profile and Add Stereotypes

Create a profile to define a set of component, port, and connection types to be used in an architecture model. For example, a profile for an electromechanical system, such as a robot, could consist of these types:

- Component types:
  - Electrical component
  - Mechanical component
  - Software component
- Connection types:
  - Analog signal connection
  - Data connection
- Port types
  - Data port

Define a profile using the Profile Editor. On the **Modeling** tab, in the **Profiles** section, select **Import**, then from the drop-down, select **Edit** . Click **New Profile**. Select new profile to start editing.

Name the profile and provide a description. Add stereotypes by clicking **New Stereotype**. You can delete stereotypes and profiles by clicking the ✖ button in their respective menus.

Save the profile. The file name is the same as the profile name.

Add Properties with Stereotypes

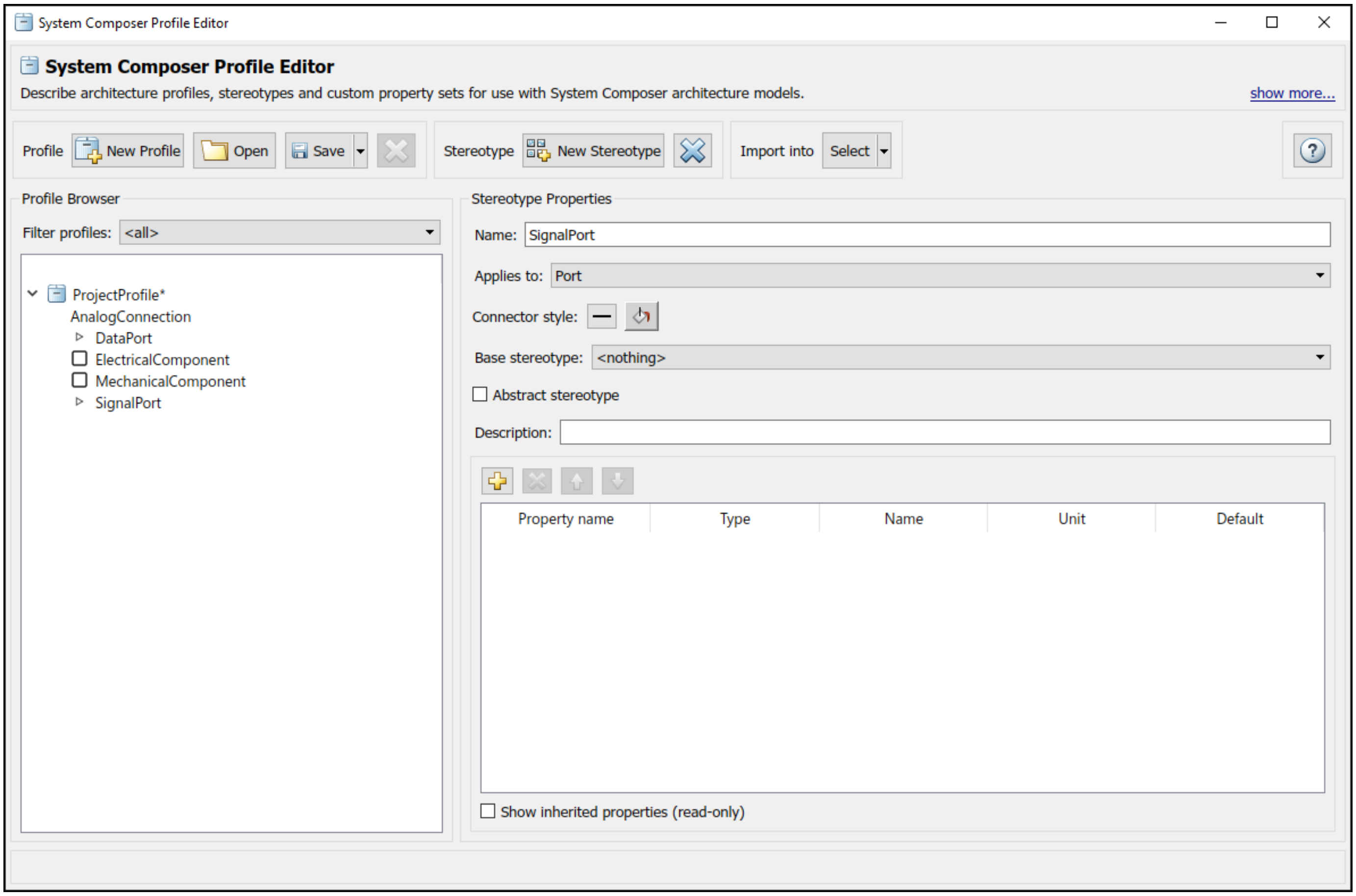
Select a stereotype in a profile to define it:

- **Name** — The name of the stereotype, for example, ElectricalComponent.
- **Applies to** — The model element type to which the stereotype applies. This field can be an <all>, component, port, connector, or interface. You can apply this stereotype only to a model element of this type.

- **Icon** — Icon to be shown on the model element with color, if applicable.
- **Connector Style** — Line style of the connector to be shown on the model with color, if applicable.
- **Base stereotype** — Other stereotype on which this stereotype is based. This can be empty.
- **Abstract stereotype** — A stereotype that is not intended to be applied directly to a model element. You can use abstract stereotypes only as the base stereotype for other stereotypes.

Add properties to a stereotype using the ➕ button. Define these fields for each property:

- Property name — Valid variable name
- Type — Numeric, string, or enumeration data type
- Name — Name of the enumerated type, if applicable
- Unit — Value units as a string
- Default — Default value



System Composer Profile Editor

Describe architecture profiles, stereotypes and custom property sets for use with System Composer architecture models.

Filter profiles: <all>

ProjectProfile*
- AnalogConnection
- DataPort
- ☐ ElectricalComponent
- ☐ MechanicalComponent
- SignalPort

Stereotype Properties

Name: SignalPort

Applies to: Port

Connector style:

Base stereotype: <nothing>

☐ Abstract stereotype

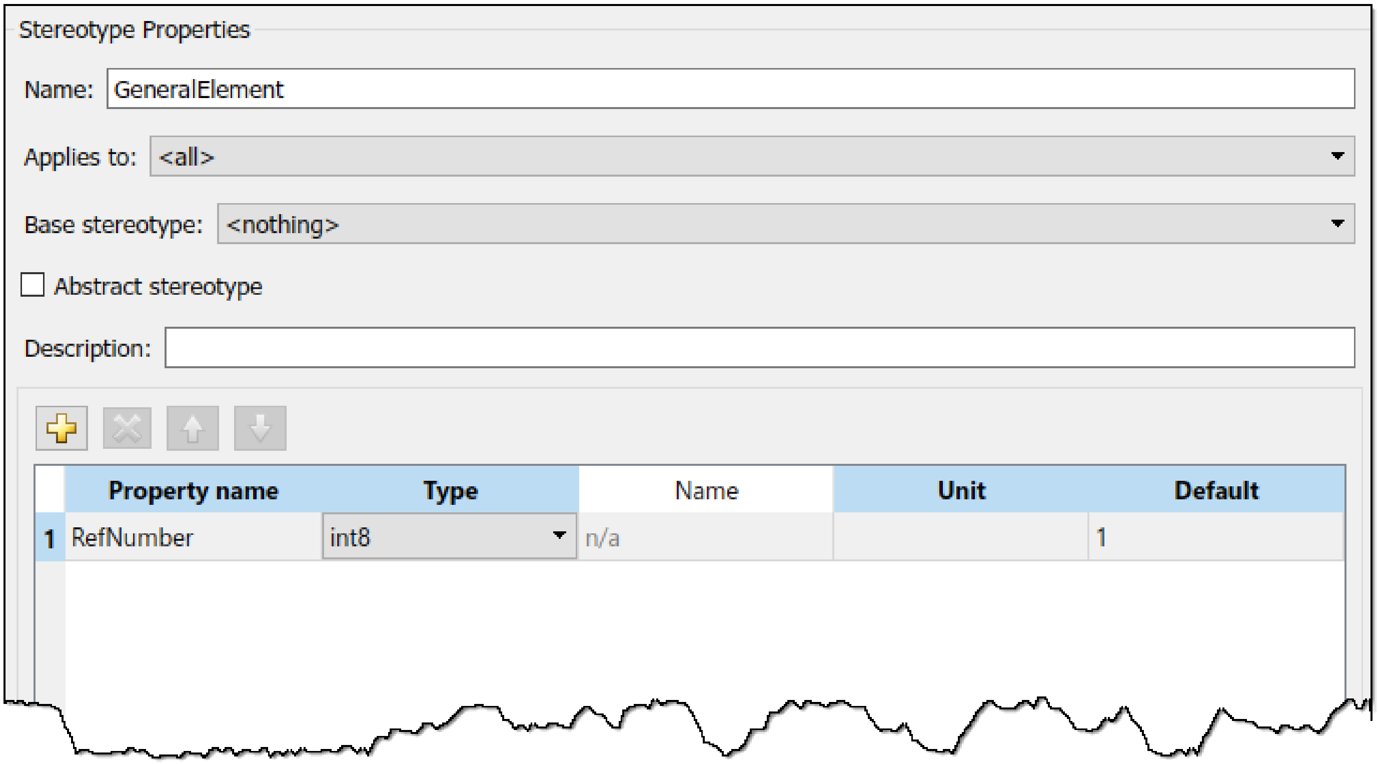
Description:

Property name	Type	Name	Unit	Default

☐ Show inherited properties (read-only)

Add, delete, and reorder properties using the property toolstrip: ➕ ✖ ⇧ ⇩

You can create a stereotype that applies to all model element types by setting the **Applies to** field to **<all>**. With these stereotypes, you can add properties to elements regardless of whether they are components, ports, connectors, or interfaces.



Stereotype Properties

Name: GeneralElement

Applies to: <all>

Base stereotype: <nothing>

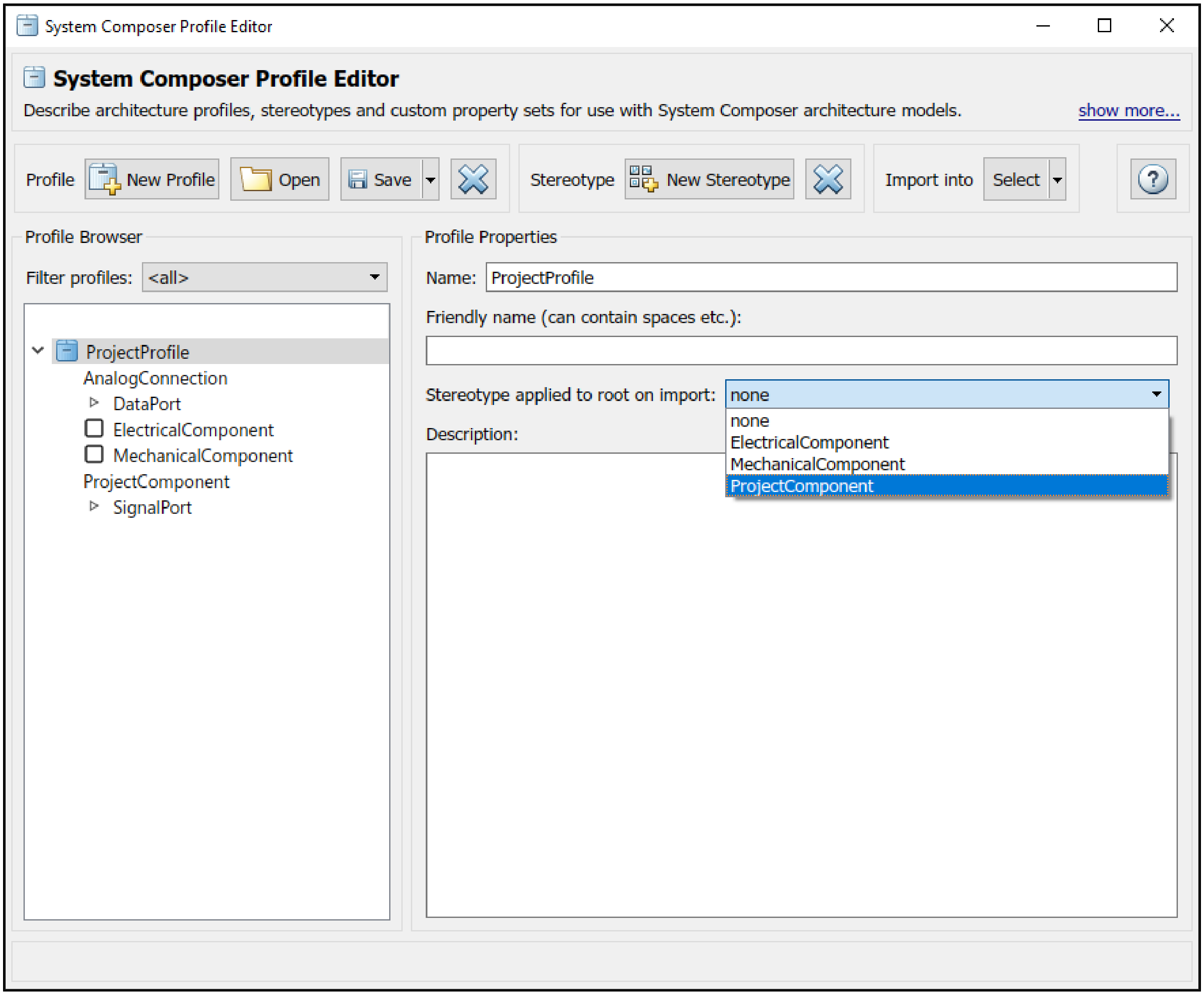
□ Abstract stereotype

Description:

	Property name	Type	Name	Unit	Default
1	RefNumber	int8	n/a		1

Default Stereotypes

Each profile can have a set of default stereotypes. Use default stereotypes when each new element of a certain type must assume the same stereotype. System Composer applies a default stereotype to the root architecture when you import the profile. You can set this default in the Profile Editor using the **Stereotype applied to root on import** field.



System Composer Profile Editor

Describe architecture profiles, stereotypes and custom property sets for use with System Composer architecture models. show more...

Filter profiles: <all>

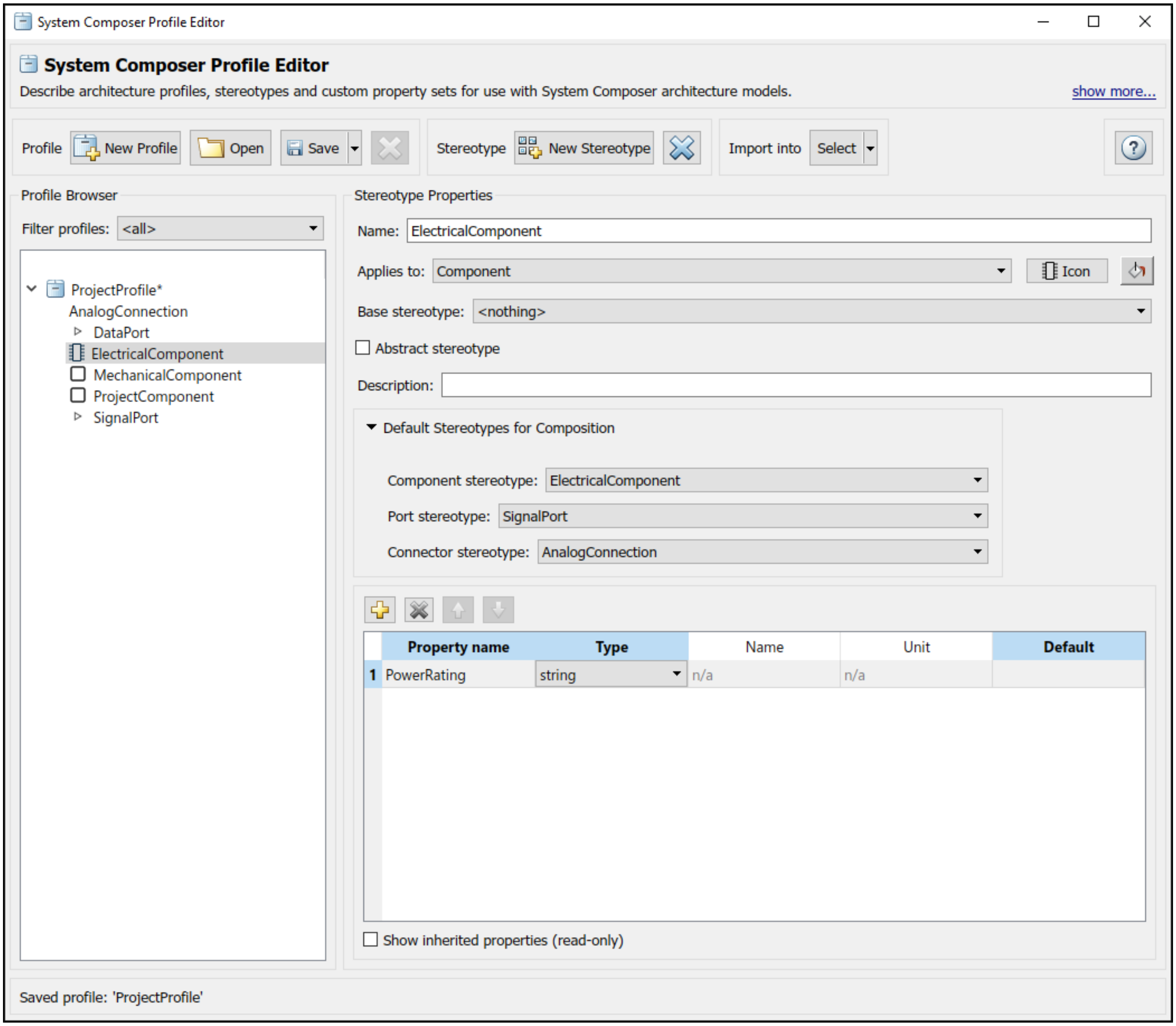
Name: ProjectProfile

Friendly name (can contain spaces etc.):

Stereotype applied to root on import: none

This default stereotype is for the top-level architecture. If a model imports multiple profiles, the default component stereotype for all profiles apply to the architecture.

Each component stereotype can also have defaults for the components, ports, and connections added to its architecture. For example, if you want all new connections in an electrical component to be analog connections, set `AnalogConnection` as a default stereotype for the `ElectricalComponent` stereotype.



System Composer Profile Editor

Describe architecture profiles, stereotypes and custom property sets for use with System Composer architecture models.

Filter profiles: <all>

- ProjectProfile*
  - AnalogConnection
  - DataPort
  - ElectricalComponent
  - MechanicalComponent
  - ProjectComponent
  - SignalPort

Name: ElectricalComponent

Applies to: Component

Base stereotype: <nothing>

☐ Abstract stereotype

Description:

Default Stereotypes for Composition

Component stereotype: ElectricalComponent

Port stereotype: SignalPort

Connector stereotype: AnalogConnection

	Property name	Type	Name	Unit	Default
1	PowerRating	string	n/a	n/a	

☐ Show inherited properties (read-only)

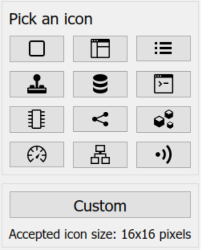
Saved profile: 'ProjectProfile'

After you import the profile into a model, all new connections assume the `AnalogConnection` stereotype.

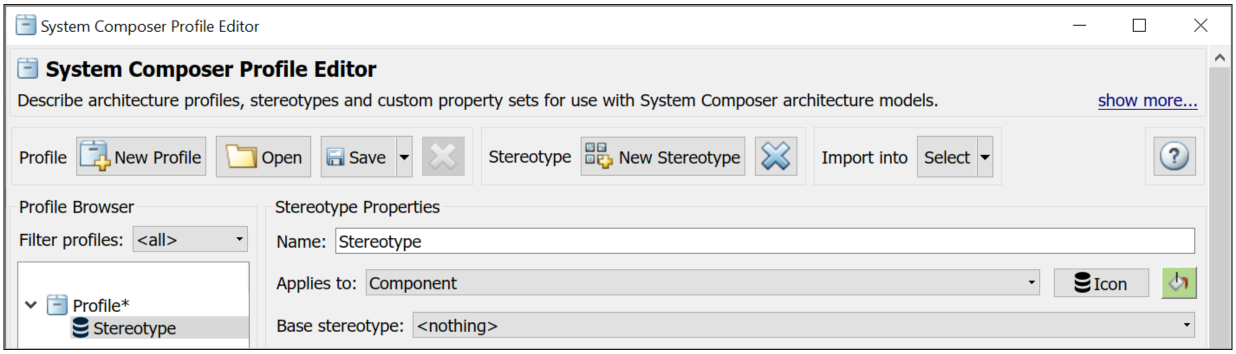
Stereotype-Based Styling

Profiles and stereotypes are used to apply custom metadata on the architecture model elements. Element styling is an additional visual cue that indicates applied stereotypes.

You can use provided icons for the component stereotypes or use you own custom icon images. Custom icons support `.png`, `.jpeg`, or `.svg` image files of size 16-by-16 pixels. The custom icons are displayed as badges on the components for which the stereotypes are applied.



You can associate a color with component stereotypes. Element styling is an additional visual cue that indicates applied stereotypes.


System Composer Profile Editor
System Composer Profile Editor
Describe architecture profiles, stereotypes and custom property sets for use with System Composer architecture models.
show more...
Profile
New Profile
Open
Save
Stereotype
New Stereotype
Import into
Select
Profile Browser
Stereotype Properties
Icon
Profile*
Stereotype

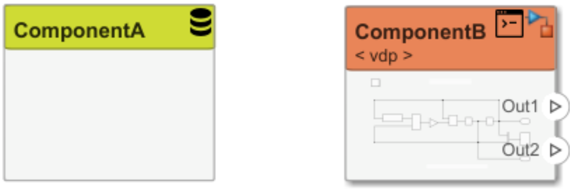
Filter profiles: <all>

Name: Stereotype

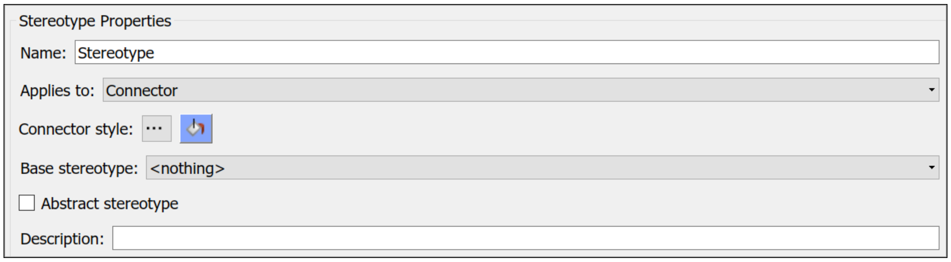
Applies to: Component

Base stereotype: <nothing>

Use a preconfigured set of color options for component stereotypes to style the architecture component headers. See “Use Stereotypes and Profiles” on page 4-10 to learn how to use stereotypes in your model.



Similarly, you can style architecture connectors using the stereotype settings. You can style connectors by using connector, port, or port interface stereotypes. Customize styling provides various color and line style choices. Connector styles are also reflected in architecture and spotlight views.



Name: Stereotype

Applies to: Connector

Connector style: ···

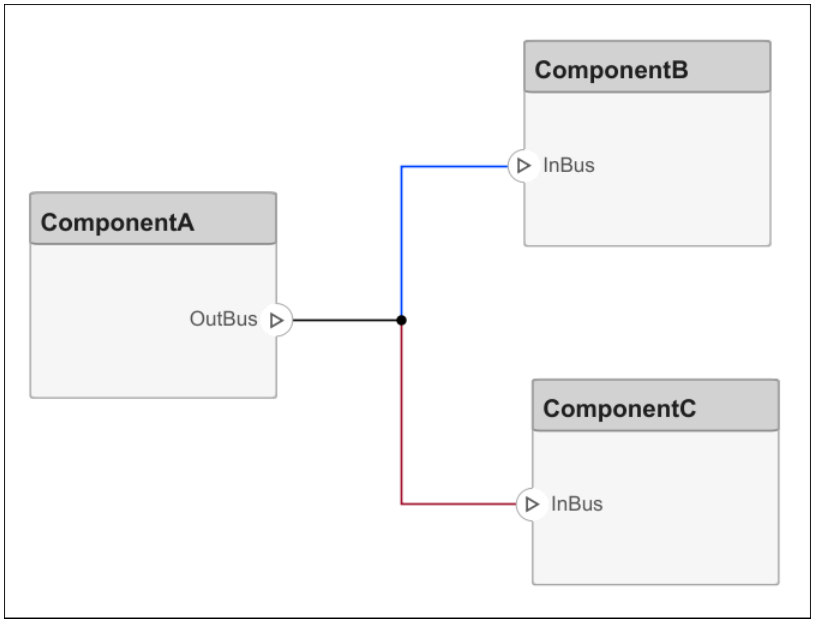
Base stereotype: <nothing>

□ Abstract stereotype

Description:

Connector styling is sourced from the highest-priority stereotype that defines style information. Connector stereotypes have the highest priority, followed by port stereotypes and then interface stereotypes.

When two connectors with different styling merge, if the styling is incompatible, the resulting connector is displayed in black.



See Also

`editor` | `systemcomposer.profile.Profile` | `systemcomposer.profile.Property` | `systemcomposer.profile.Stereotype`

More About

- “Use Stereotypes and Profiles” on page 4-10

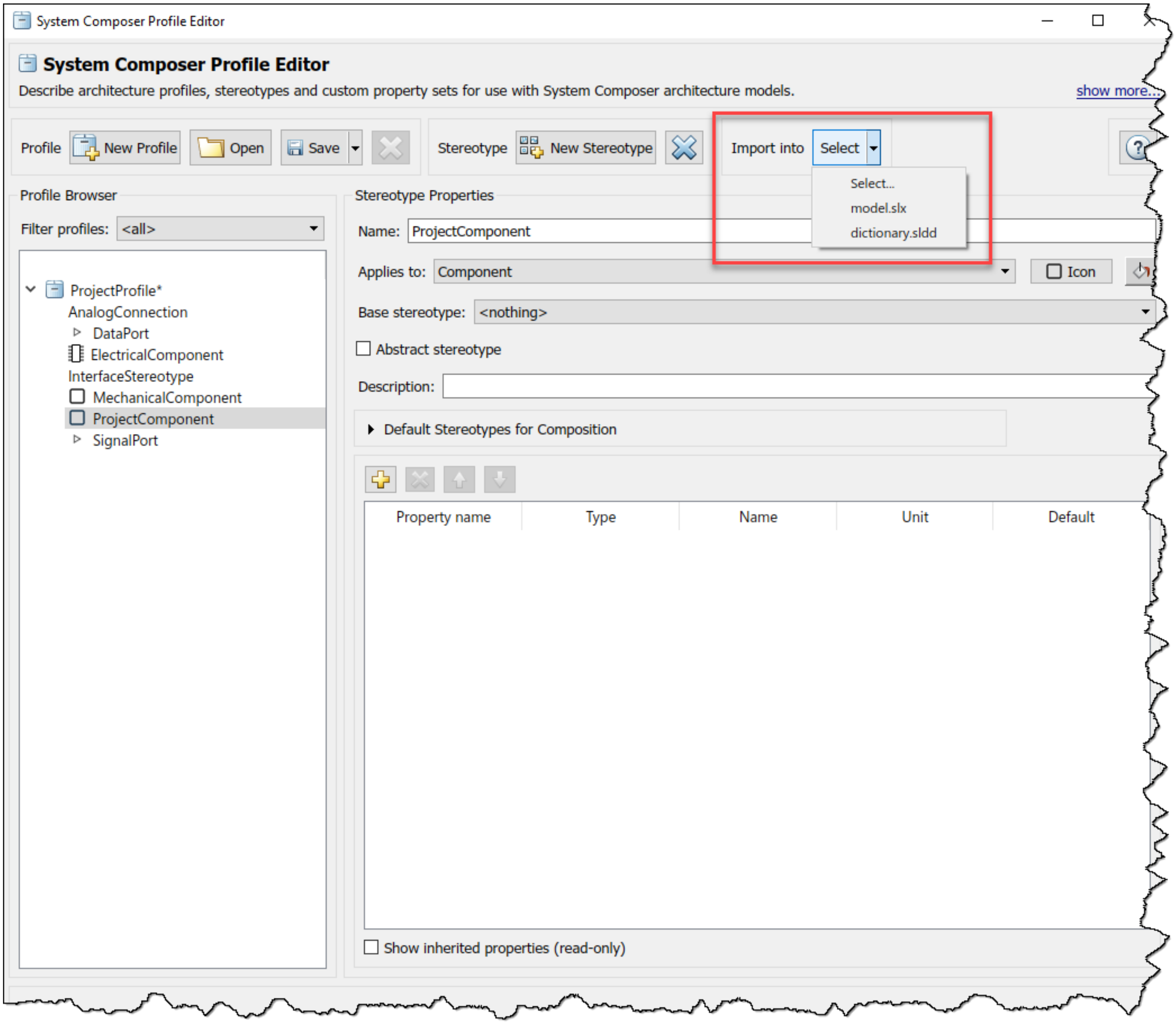
Use Stereotypes and Profiles

Use profiles to add properties to components, ports, and connectors. Import an existing profile, apply stereotypes, and add property values. To create a profile, see “Define Profiles and Stereotypes” on page 4-2.

Import Profiles

The Profile Editor is independent from the model that opens it, so you must explicitly import a new profile into a model. The profile must first be saved with an `.xml` extension. On the **Modeling** tab, in the **Profiles** section, select **Import**, then from the drop-down, select **Import**. Select the profile to import. An architecture model can use multiple profiles at once.

Alternatively, open the Profile Editor. On the **Modeling** tab, in the **Profiles** section, select **Import**, then from the drop-down, select **Edit**. You can import a profile into any open dictionaries or models.



Filter profiles: <all>

Name: ProjectComponent

Applies to: Component

Base stereotype: <nothing>

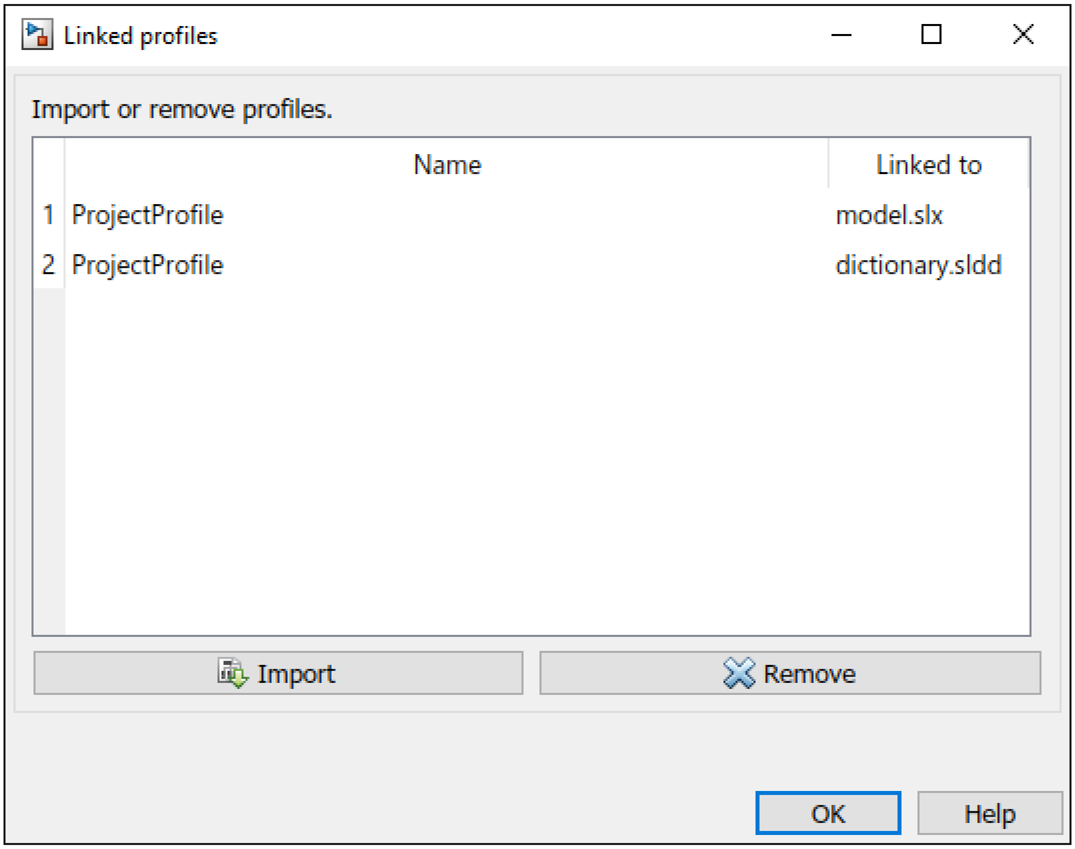
□ Abstract stereotype

Description:

□ Show inherited properties (read-only)

Note For a System Composer component that is linked to a Simulink behavior model, the profile must be imported into the Simulink model before applying a stereotype from it to the component. Since the Property Inspector on the Simulink side does not display stereotypes, this workflow is not finalized.

To manage profiles after they have been imported, in the **Profiles** section, select **Import**, then from the drop-down, select **Manage** .

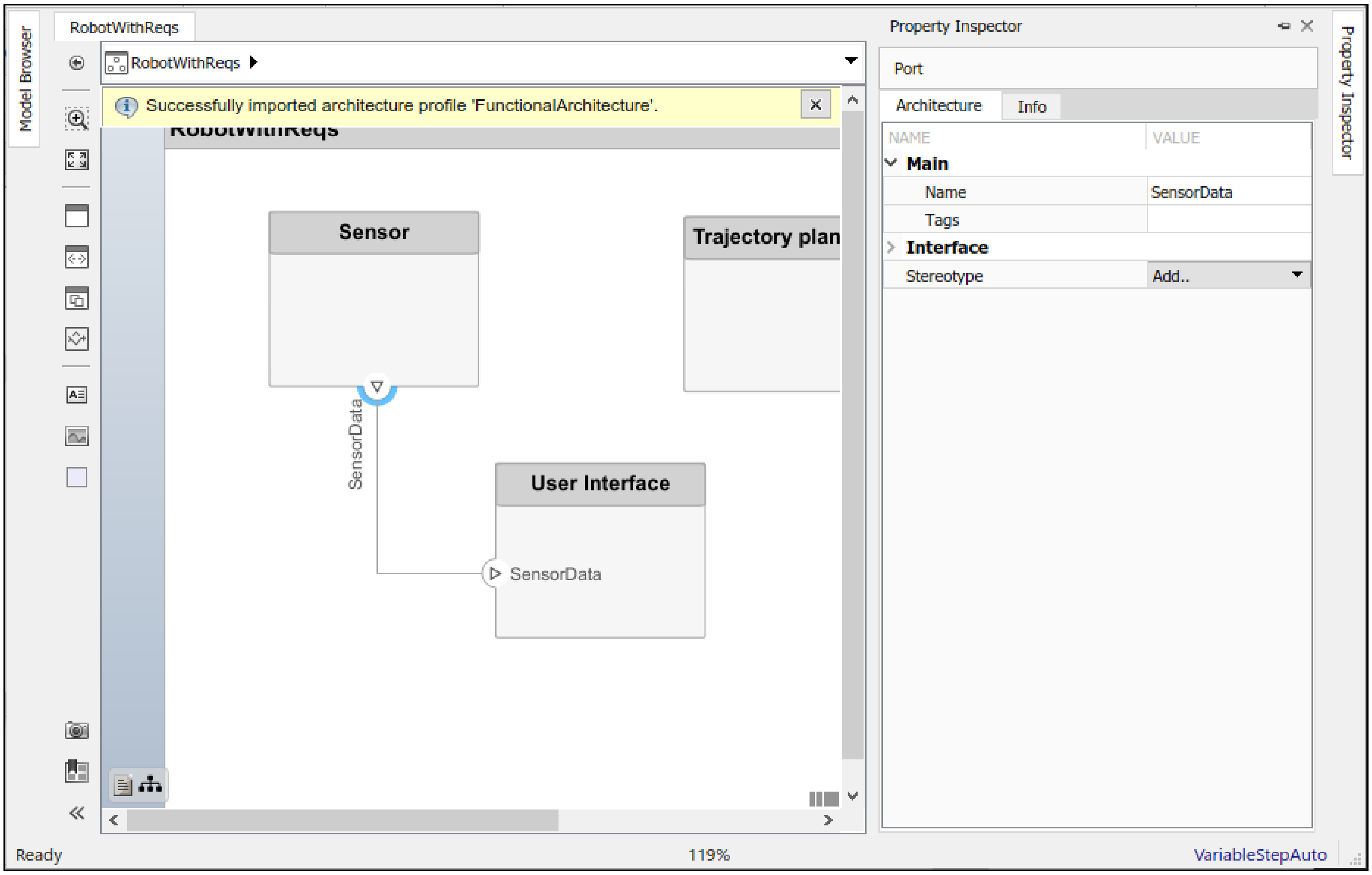


Import or remove profiles.

	Name	Linked to
1	ProjectProfile	model.slx
2	ProjectProfile	dictionary.sldd

Apply a Stereotype

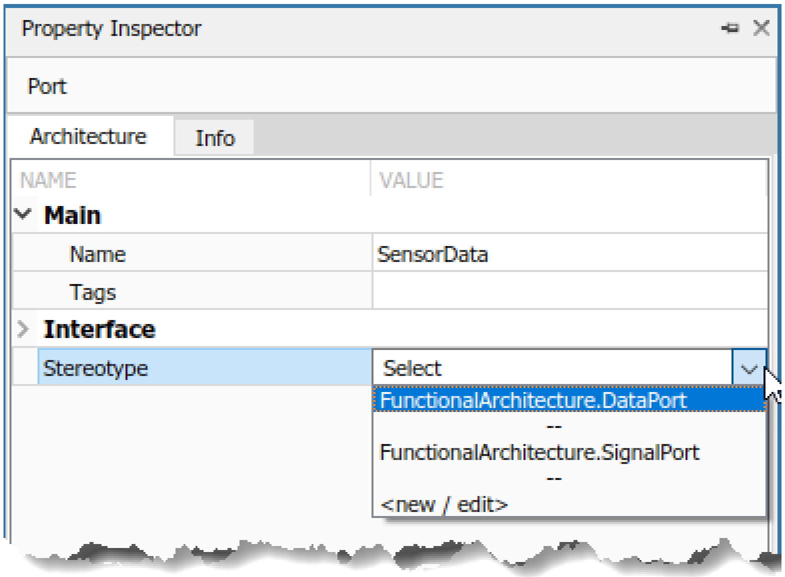
Once the profile is available in the model, open the Property Inspector. On the **Modeling** tab, in the **Design** section, select **Property Inspector**. Select a model element.



Successfully imported architecture profile 'FunctionalArchitecture'.

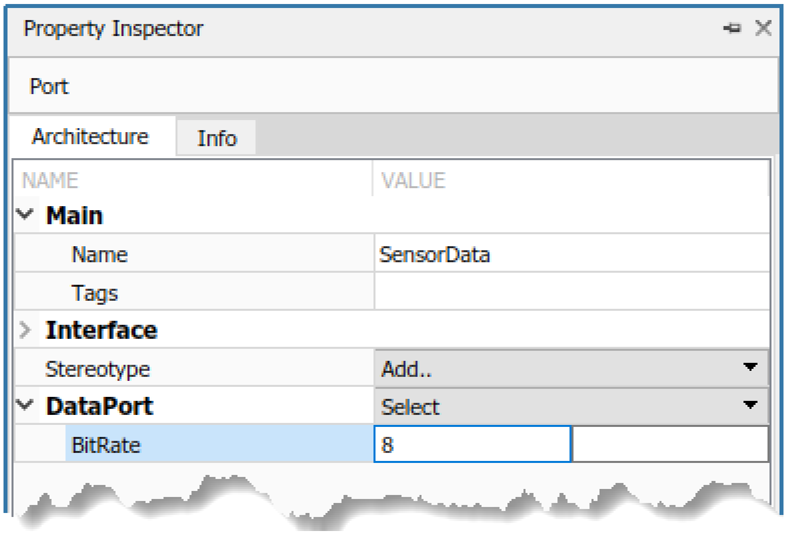
NAME	VALUE
Name	SensorData
Tags	
Stereotype	Add..

In the **Stereotype** field, use the drop-down to select the stereotype. Only the stereotypes that apply to the current element type (for example, a port) are available for selection. If no stereotype exists, you can use the **<new / edit>** option to open the Profile Editor and create one.



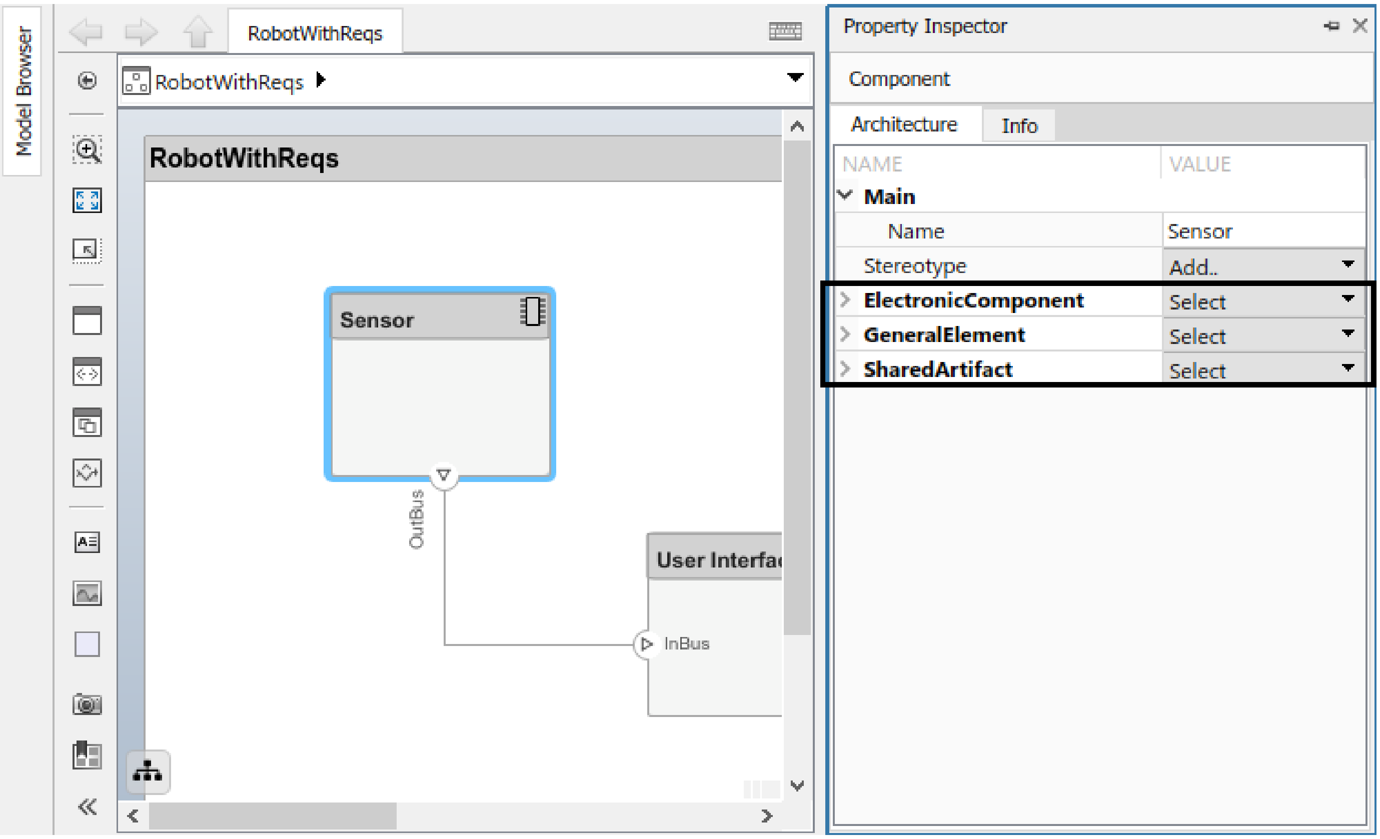
NAME	VALUE
Name	SensorData
Tags	
Stereotype	Select

When you apply a stereotype to an element, a new set of properties appears in the Property Inspector under the name of the stereotype. To edit the properties, expand this set.



NAME	VALUE
Main	
Name	SensorData
Tags	
Interface	
Stereotype	Add..
DataPort	Select
BitRate	8

You can set multiple stereotypes for each element.

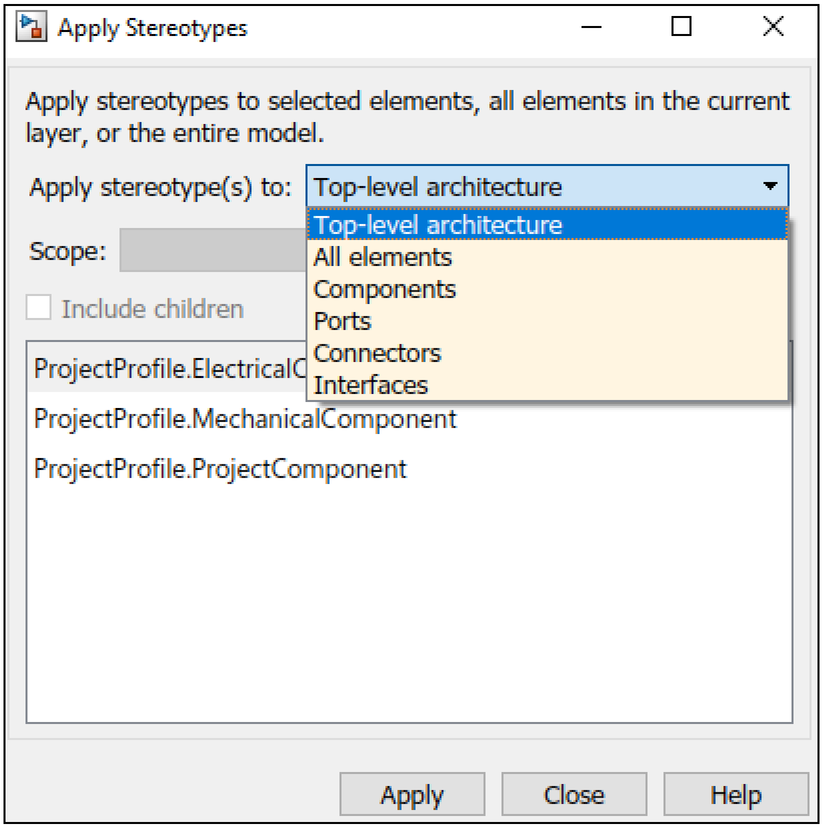


NAME	VALUE
Main	
Name	Sensor
Stereotype	Add..
ElectronicComponent	Select
GeneralElement	Select
SharedArtifact	Select

You can also apply component, port, connector, and interface stereotypes to all applicable elements at the same architecture level. On the **Modeling** tab, in the **Profiles** section, select **Apply Stereotypes**.

In the Apply Stereotypes dialog box, from **Apply stereotype(s) to**, select `Top-level architecture`, `All elements`, `Components`, `Ports`, `Connectors`, or `Interfaces`.

Note The `Interfaces` option is only available if interfaces are defined in the Interface Editor. For more information, see “Define Interfaces” on page 3-2.



Apply Stereotypes

Apply stereotypes to selected elements, all elements in the current layer, or the entire model.

Apply stereotype(s) to: Top-level architecture

Scope:

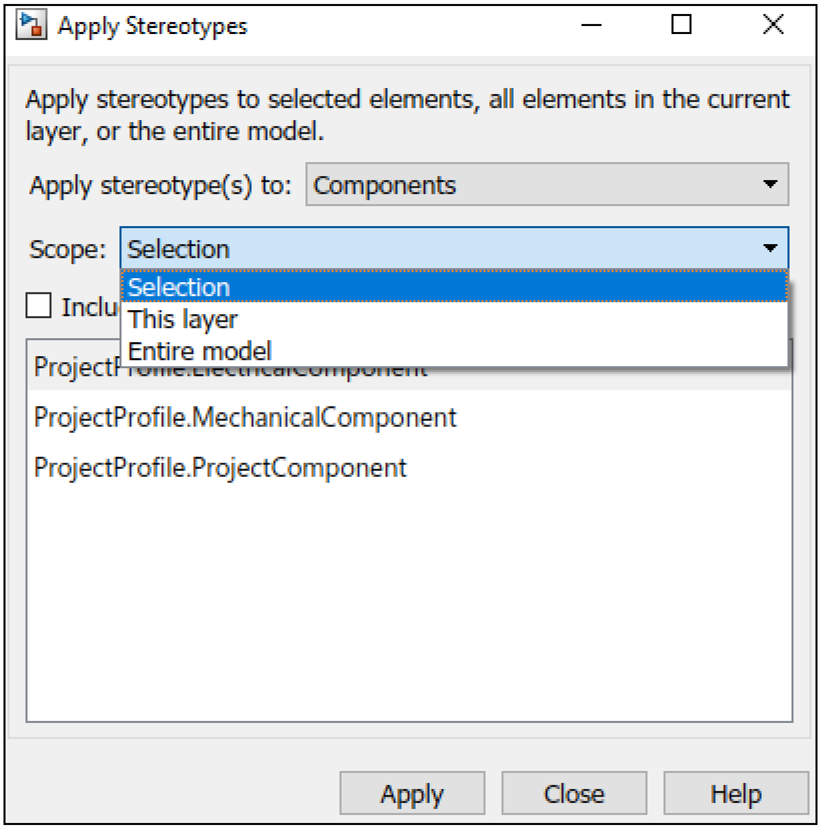
□ Include children

ProjectProfile.ElectricalC

ProjectProfile.MechanicalComponent

ProjectProfile.ProjectComponent

You can also apply stereotypes by selecting a single model element. From the **Scope** list, select `Selection`, `This layer`, or `Entire model`.



Apply Stereotypes

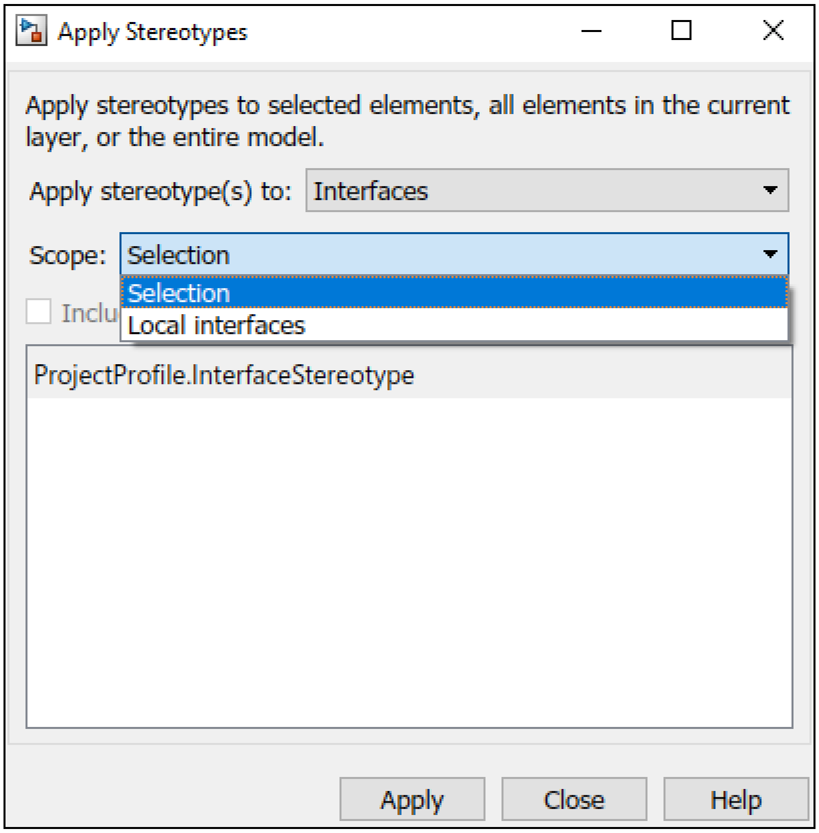
Apply stereotypes to selected elements, all elements in the current layer, or the entire model.

Apply stereotype(s) to: Components

Scope: Selection

Selection
This layer
Entire model
□ Inclu
ProjectProfile.ElectricalComponent
ProjectProfile.MechanicalComponent
ProjectProfile.ProjectComponent
Apply
Close
Help

You can also apply stereotypes to interfaces. When interfaces are locally defined and you select one or more interfaces in the Interface Editor, the options for **Scope** are `Selection` and `Local interfaces`.



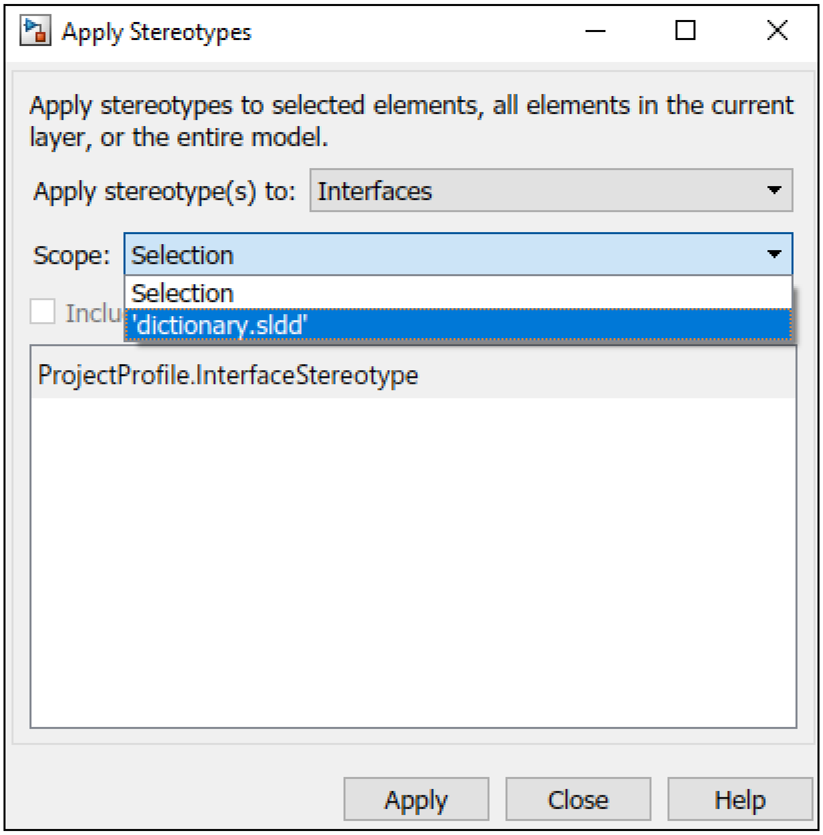
Apply stereotypes to selected elements, all elements in the current layer, or the entire model.

Apply stereotype(s) to: Interfaces

Scope: Selection

ProjectProfile.InterfaceStereotype

When interfaces are stored and shared across a data dictionary and you select one or more interfaces in the Interface Editor, the options for **Scope** are `Selection` and either `dictionary.sldd` or the name of the dictionary currently in use.



Apply Stereotypes

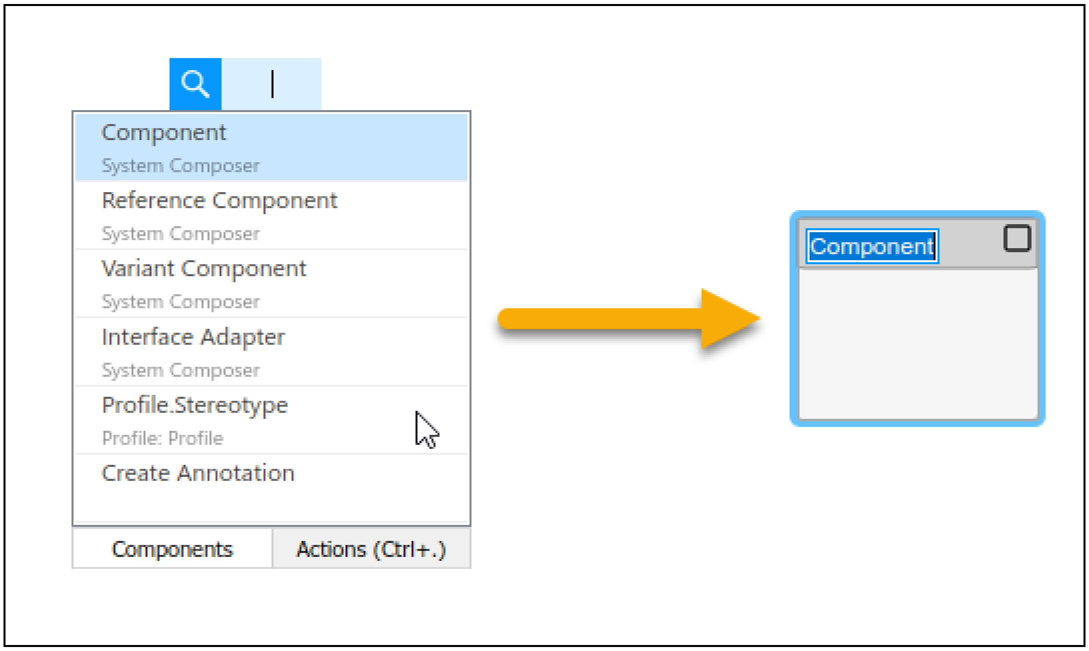
Apply stereotypes to selected elements, all elements in the current layer, or the entire model.

Apply stereotype(s) to: Interfaces

Scope: Selection

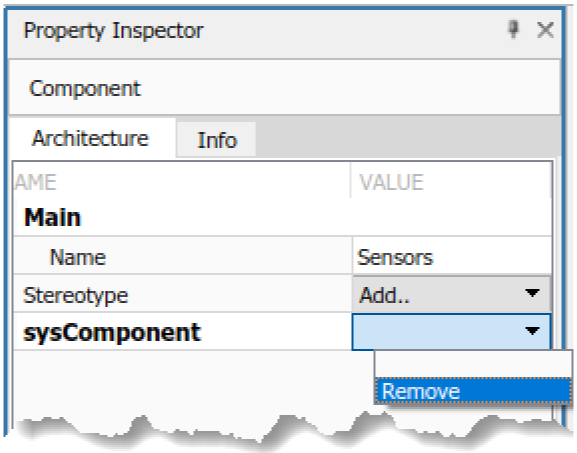
Note For the stereotypes to display for interfaces in a dictionary, in the Apply Stereotypes dialog box, the profile must be imported into the dictionary.

You can also create a new component with an applied stereotype using the quick-insert menu. Select the stereotype as a fully qualified name. A component with that stereotype is created.



Remove a Stereotype

If a stereotype is no longer required for an element, remove it using the Property Inspector. Click **Select** next to the stereotype and choose **Remove**.



Name: Sensors

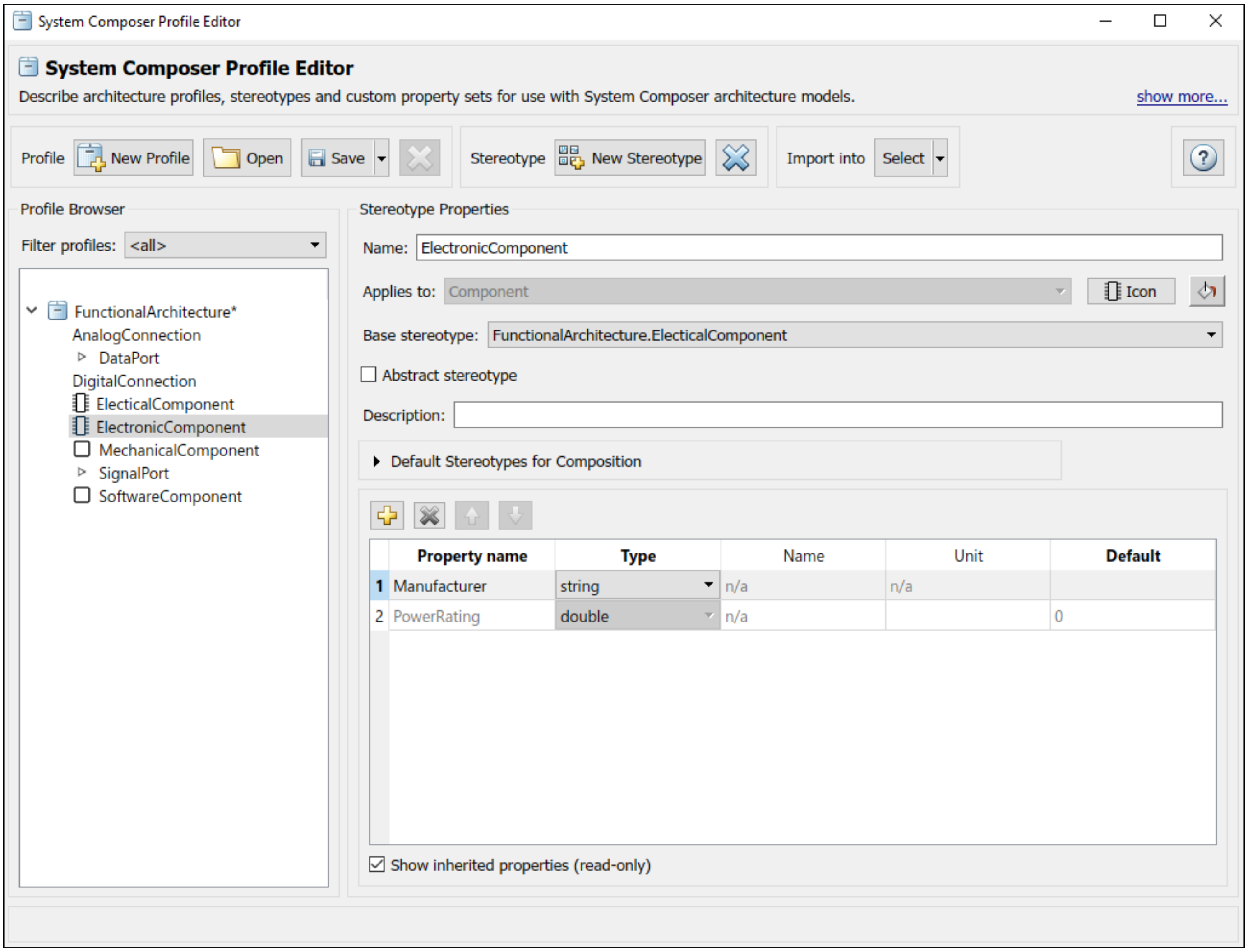
Stereotype: Add..

sysComponent

Extend a Stereotype

You can extend a stereotype by creating a new stereotype based on the existing one, allowing you to control properties in a structural manner. For example, all components in a project may have a part number, but only electrical components have a power rating, and only electronic components — a subset of electrical components — have manufacturer information. You can use an abstract stereotype to serve solely as a base for other stereotypes and not as a stereotype for any architecture model elements.

For example, create a new stereotype called `ElectronicComponent` in the Profile Editor. Select its base stereotype as `FunctionalArchitecture.ElectricalComponent`. Define properties you are adding to those of the base stereotype. Check **Show inherited properties** at the bottom of the property list to show the properties of the base stereotype. You can edit only the properties of the selected stereotype, not the base stereotype.



System Composer Profile Editor

Describe architecture profiles, stereotypes and custom property sets for use with System Composer architecture models.

Profile Browser

Filter profiles: <all>

- FunctionalArchitecture*
  - AnalogConnection
  - DataPort
  - DigitalConnection
  - ElecticalComponent
  - ElectronicComponent
  - MechanicalComponent
  - SignalPort
  - SoftwareComponent

Stereotype Properties

Name: ElectronicComponent

Applies to: Component

Base stereotype: FunctionalArchitecture.ElecticalComponent

□ Abstract stereotype

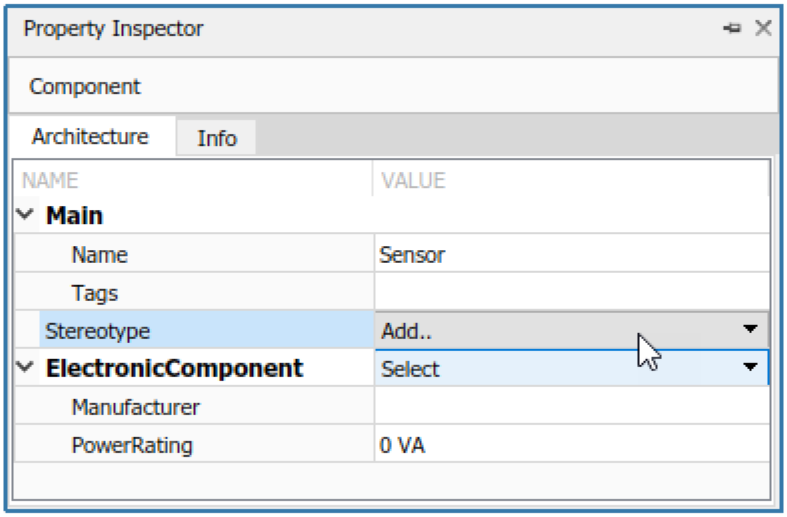
Description:

Default Stereotypes for Composition

	Property name	Type	Name	Unit	Default
1	Manufacturer	string	n/a	n/a	
2	PowerRating	double	n/a		0

☑ Show inherited properties (read-only)

When you apply the new stereotype, it carries its defined properties in addition to those of its base stereotype.



NAME	VALUE
Main	
Name	Sensor
Tags	
Stereotype	Add..
ElectronicComponent	Select
Manufacturer	
PowerRating	0 VA

See Also

`editor` | `systemcomposer.profile.Profile` | `systemcomposer.profile.Property` | `systemcomposer.profile.Stereotype`

More About

- “Define Profiles and Stereotypes” on page 4-2
- “Analyze Architecture” on page 6-10

Use Simulink Models with System Composer

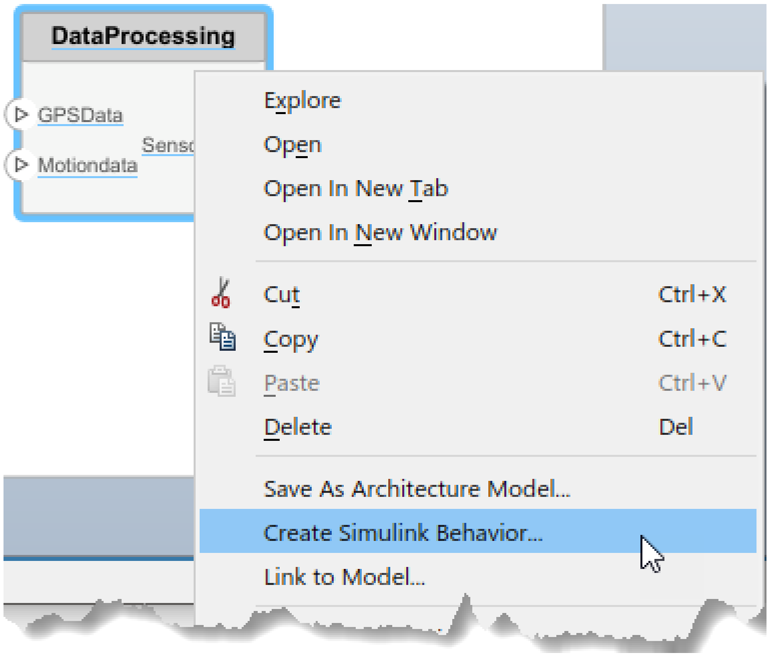
Implement Component Behavior in Simulink

System design and architecture definition can involve a behavior definition for some components, such as the algorithm for a data processing component. Components in System Composer architecture models can define behavior using Simulink models by linking components to Simulink models.

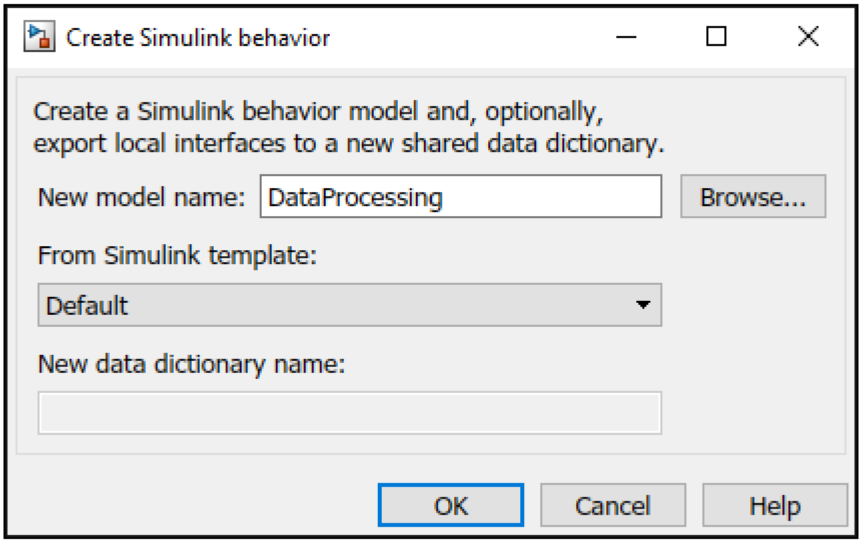
Create a Simulink Behavior Model

When a component does not require further decomposition from an architecture standpoint, you can design and define its behavior in Simulink. When linked to a Simulink behavior, the component becomes a Reference Component. A reference component represents a logical hierarchy of other compositions. You can reuse compositions in the model using reference components.

1 Right-click the component and select `Create Simulink Behavior`, or, on the toolstrip under **Component**, click **Create Simulink Behavior**.



2 Provide a model name. The default name is the name of the component.



Create Simulink behavior

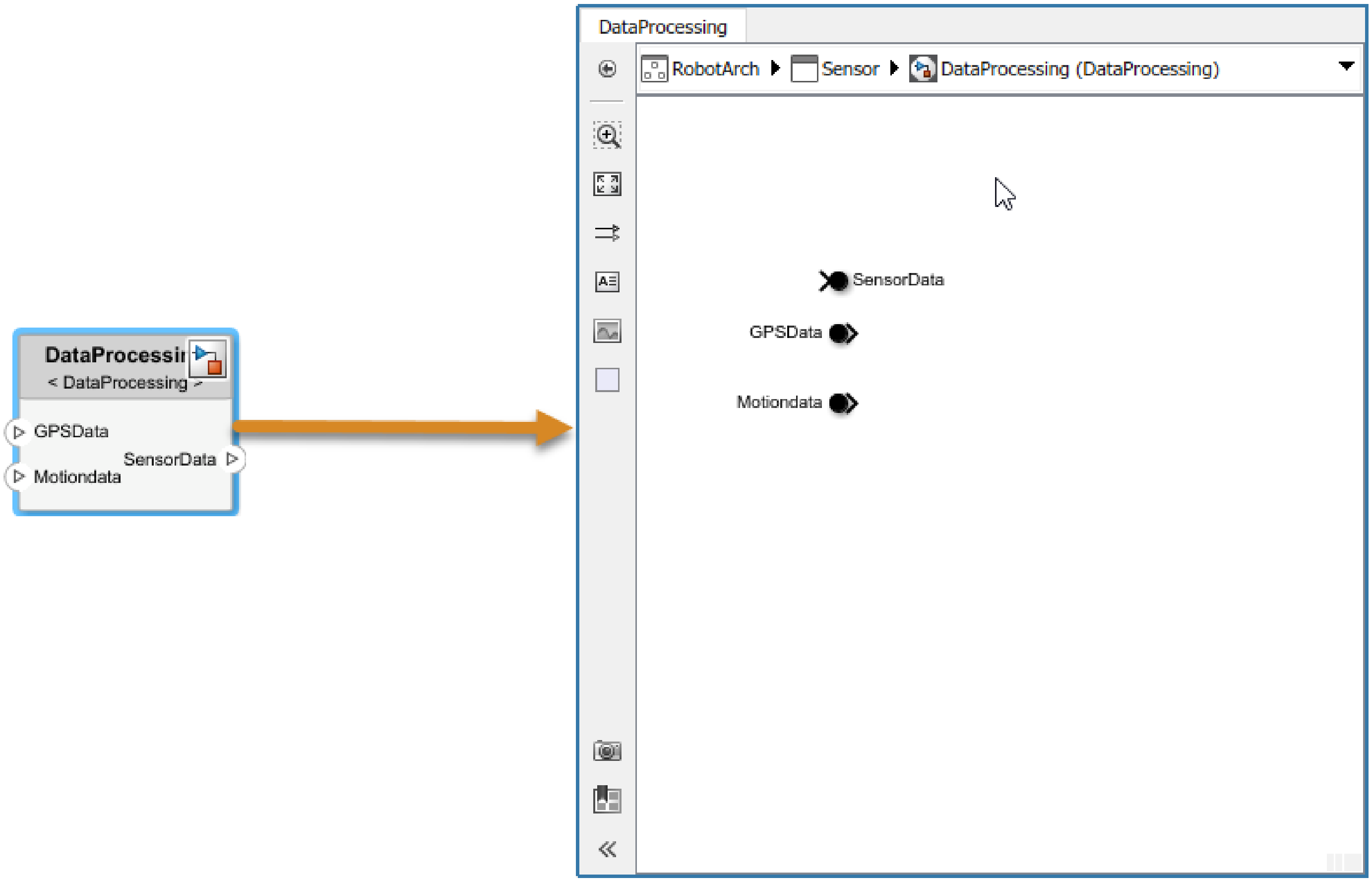
Create a Simulink behavior model and, optionally, export local interfaces to a new shared data dictionary.

New model name: DataProcessing

From Simulink template: Default

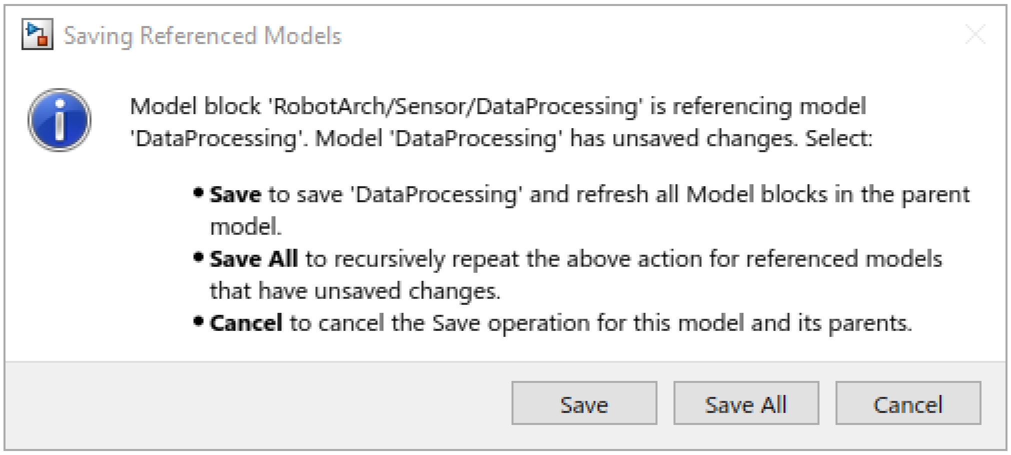
New data dictionary name:

- A new Simulink model with the provided name is created. The root level ports of the Simulink model reflect the ports of the component.
- The component in the architecture model is linked to the Simulink model. The Simulink icon on the component indicates this is a Simulink link.



You can continue with providing specific dynamics and algorithms in the referenced Simulink model. Adding root-level ports in the Simulink model creates additional ports on the System Composer Reference Component block.

You can access and edit a referenced Simulink model by double-clicking the component in the architecture model. When you save the architecture model, all unsaved Simulink behavior models it references must also be saved, and all linked components updated.

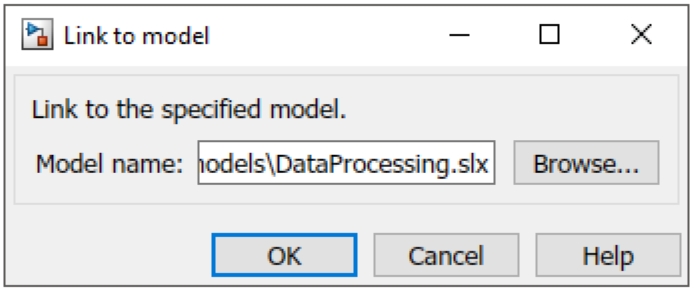


Model block 'RobotArch/Sensor/DataProcessing' is referencing model 'DataProcessing'. Model 'DataProcessing' has unsaved changes. Select:

- **Save** to save 'DataProcessing' and refresh all Model blocks in the parent model.
- **Save All** to recursively repeat the above action for referenced models that have unsaved changes.
- **Cancel** to cancel the Save operation for this model and its parents.

Link to an Existing Simulink Behavior Model

You can link to an existing Simulink behavior model from a System Composer component, provided that the component is not already linked to a reference architecture. Right-click the component and select **Link to Model**. Type in or browse for the name of a Simulink model.



Link to the specified model.

Model name: ıodels\DataProcessing.slx

Any subcomponents and ports that are present in the components get deleted when the component links to a Simulink model, with a prompt to continue and lose subcomponents and ports when linking.

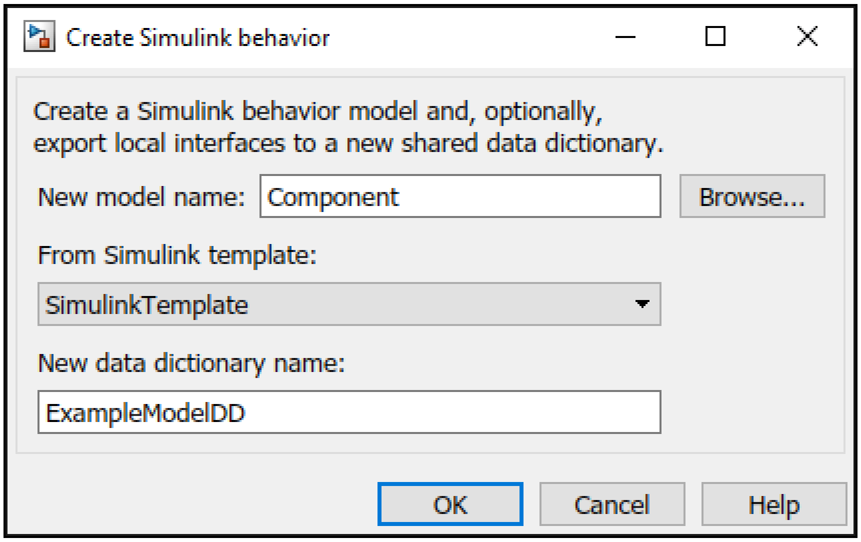
Note The software does not support linking a System Composer component to a Simulink model with root-level enable or trigger ports.

You can link protected Simulink models (`.slxp`) to create component behaviors. You can also convert an already linked Simulink behavior model to a protected model, and the change is reflected after refreshing the model.

Create a Simulink Behavior from Template for a Component

To create user-defined templates for Simulink models, see “Create Template from Model”.

After creating and saving a user-defined template, you can link the template to a Simulink behavior. Right-click the component and select `Create Simulink Behavior`, or, on the toolstrip under **Component**, click **Create Simulink Behavior**.



Create Simulink behavior

Create a Simulink behavior model and, optionally, export local interfaces to a new shared data dictionary.

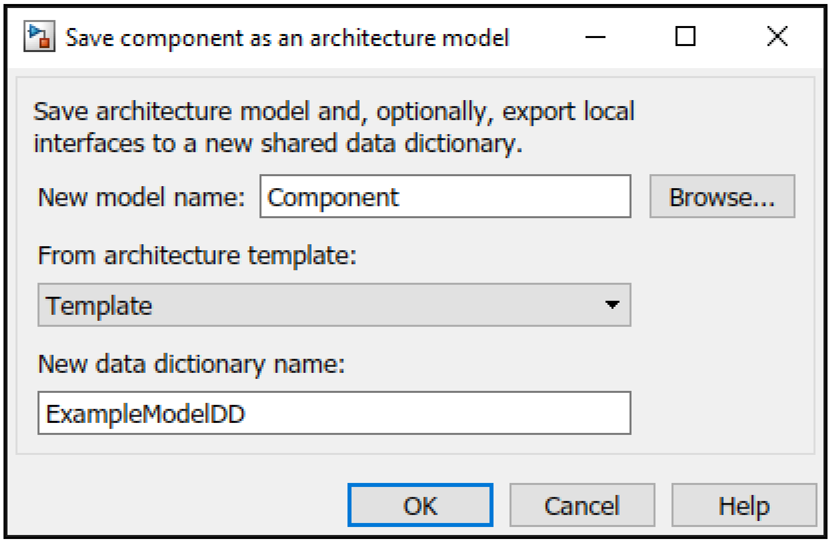
New model name: Component

From Simulink template: SimulinkTemplate

New data dictionary name: ExampleModelDD

On the **Create Simulink behavior** dialog, choose the template and enter a new data dictionary name if local interfaces are defined. Click **OK**. The component exhibits a Simulink behavior according to the template with shared interfaces, if present. Blocks and lines in the template are excluded, and only configuration settings are preserved. Configuration settings include annotations and styling.

Note Architecture templates can be used with **Save As Architecture Model**.



Save component as an architecture model

Save architecture model and, optionally, export local interfaces to a new shared data dictionary.

New model name: Component

From architecture template: Template

New data dictionary name: ExampleModelDD

See Also

Functions
`createSimulinkBehavior` | `linkToModel` | `saveAsModel`

Blocks
Reference Component

More About

- "Decompose and Reuse Components" on page 1-16
- "Add Stateflow Chart Behavior to Architecture Component" on page 5-7
- "Extract Architecture from Simulink Model" on page 5-12
- "Simulating Mobile Robot with System Composer Workflow" on page 6-46

Add Stateflow Chart Behavior to Architecture Component

A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states. Add Stateflow Chart behavior to describe a System Composer architectural component using state machines.

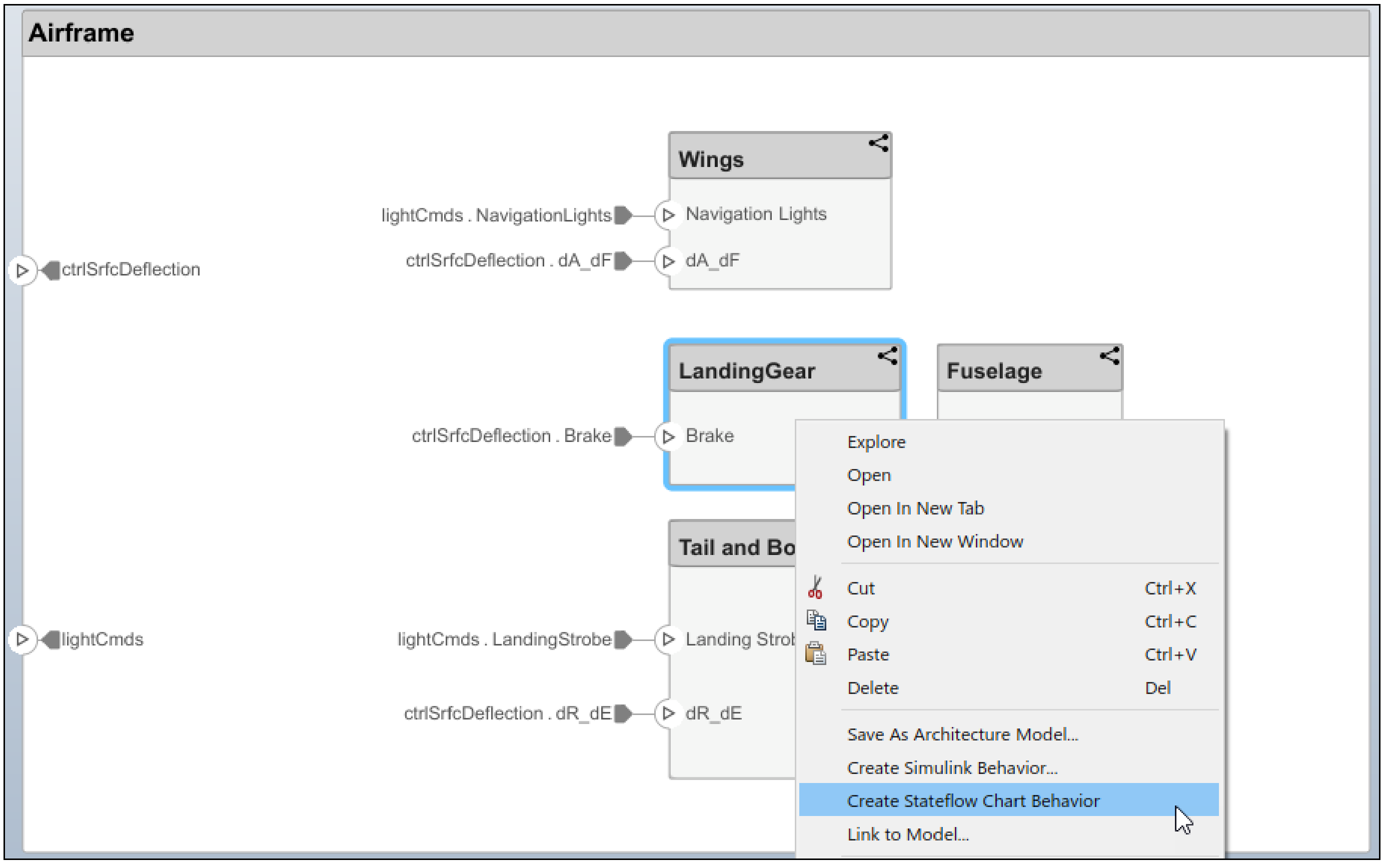
State charts consist of a finite set of states with transitions between them to capture the modes of operation for the component. Charts allow design for different modes, internal states, and event-based logic of a system. You can also use charts as stubs to mock a complex component implementation during top-down integration testing. A Stateflow license is required to use this functionality. For more information, see “Stateflow”.

Add State Chart Behavior to a Component

A System Composer component with stereotypes, interfaces, requirement links, and ports, is preserved when you add Stateflow Chart behavior.

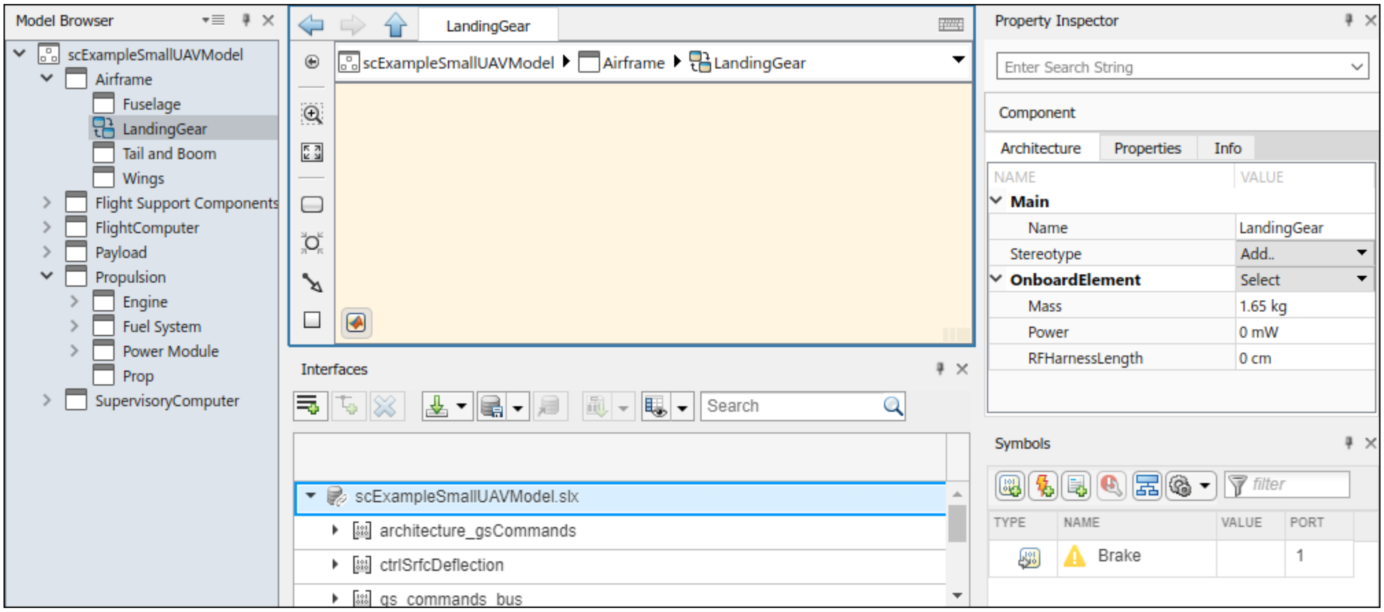
1. This example uses the architecture model for an unmanned aerial vehicle (UAV) to add state chart behavior to a component. Enter the following command:

   ```
   scExampleSmallUAV
   ```
2. Double-click the `Airframe` component. Select the `LandingGear` component on the System Composer composition editor.
3. Select the `Brake` port. Open the Interface Editor from the toolstrip **Design > Interface Editor**. Right-click the interface `operatorCmds` and select **Assign to Selected Port(s)**.
4. Right-click the `LandingGear` component and select `Create Stateflow Chart Behavior`. Alternatively, on the toolstrip, under **Component**, click **Create Stateflow Chart Behavior**.


Airframe
Wings
lightCmds . NavigationLights
Navigation Lights
ctrlSrfcDeflection . dA_dF
dA_dF
ctrlSrfcDeflection
LandingGear
Fuselage
ctrlSrfcDeflection . Brake
Brake
Tail and Bo
lightCmds
lightCmds . LandingStrobe
Landing Strol
ctrlSrfcDeflection . dR_dE
dR_dE
Explore
Open
Open In New Tab
Open In New Window
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
Save As Architecture Model...
Create Simulink Behavior...
Create Stateflow Chart Behavior
Link to Model...

5 Double-click `LandingGear`, which has the Stateflow icon. In the **Modeling** menu, select **Design Data**, then click **Symbols Pane** to view the Stateflow symbols. The input port `Brake` appears as input data in the Symbols pane.

Note Some Stateflow objects remain local to Stateflow Charts. Input and output event ports are not supported in System Composer. Only local events are supported.

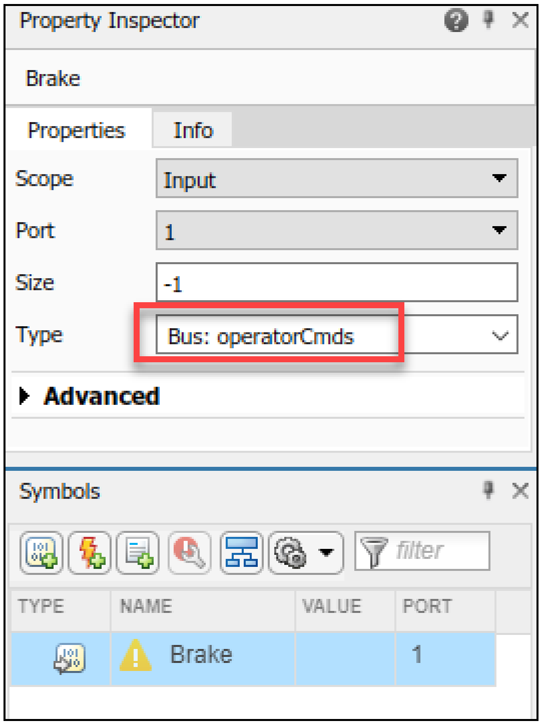


NAME	VALUE
Main	
Name	LandingGear
Stereotype	Add..
OnboardElement	Select
Mass	1.65 kg
Power	0 mW
RFHarnessLength	0 cm

TYPE	NAME	VALUE	PORT
	⚠ Brake		1

Since Stateflow ports show up as input and output data objects, they must follow Stateflow naming conventions. Ports are automatically renamed to follow Stateflow naming conventions. For more information, see “Guidelines for Naming Stateflow Objects” (Stateflow).

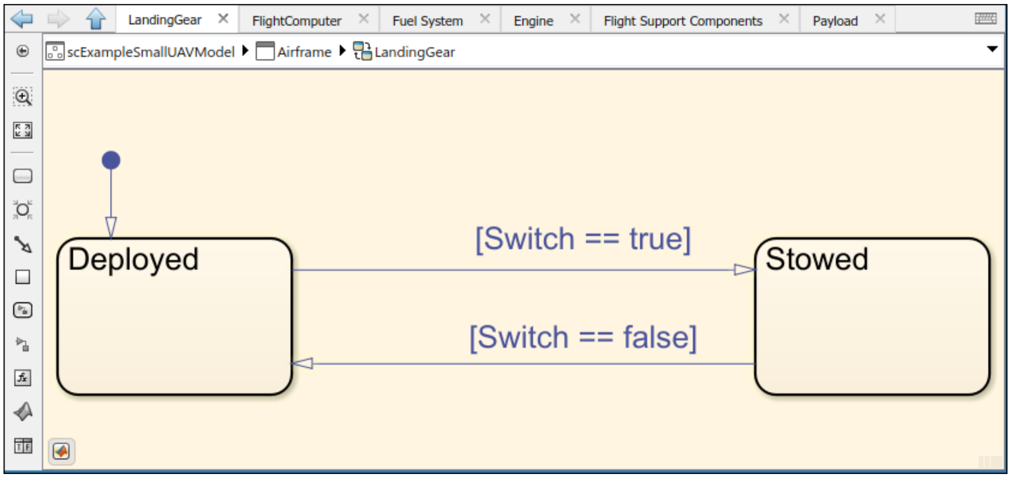
6 Select the `Brake` input and view the interface in the Property Inspector. The interface can be accessed like a Simulink bus signal. For information on how to use bus signals in Stateflow, see “Index and Assign Values to Stateflow Structures” (Stateflow).



Scope	Input
Port	1
Size	-1
Type	Bus: operatorCmds

TYPE	NAME	VALUE	PORT
	⚠ Brake		1

7 You can populate the Stateflow canvas to represent the internal states of the `LandingGear`.

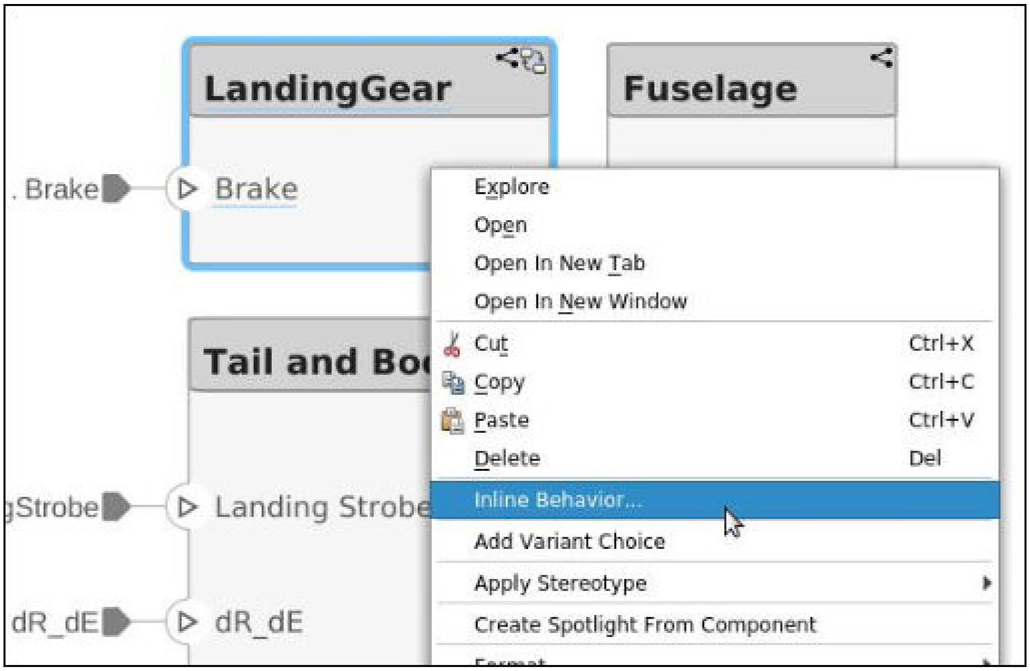


A Stateflow Chart behavior added to a component is part of the same System Composer architecture model that contains the component.

Inline Stateflow Chart Behavior

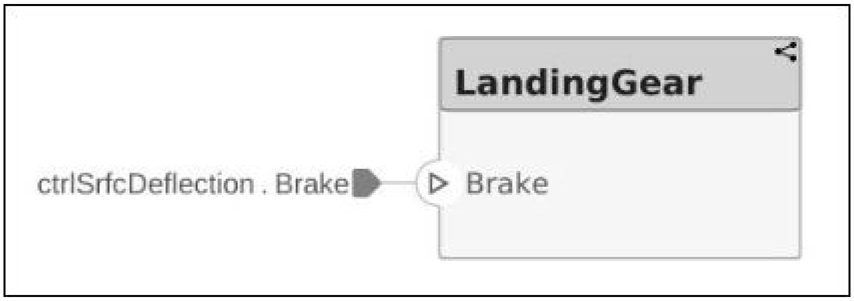
You can inline a component with a Stateflow Chart behavior to delete the contents inside the Stateflow Chart while preserving interfaces.

1 Right-click on the `LandingGear` component and select `Inline Behavior`.



2 To confirm the operation to delete all the content inside the Stateflow Chart behavior, click **OK**.

3 The Stateflow Chart behavior on the component is removed and the component is inlined with interfaces.



See Also

`createStateflowChartBehavior` | `inlineComponent`

More About

- “Decompose and Reuse Components” on page 1-16
- “Implement Component Behavior in Simulink” on page 5-2
- “Extract Architecture from Simulink Model” on page 5-12
- “Define Sequence Diagrams” on page 5-16
- “Use Sequence Diagrams in the Views Gallery” on page 5-27

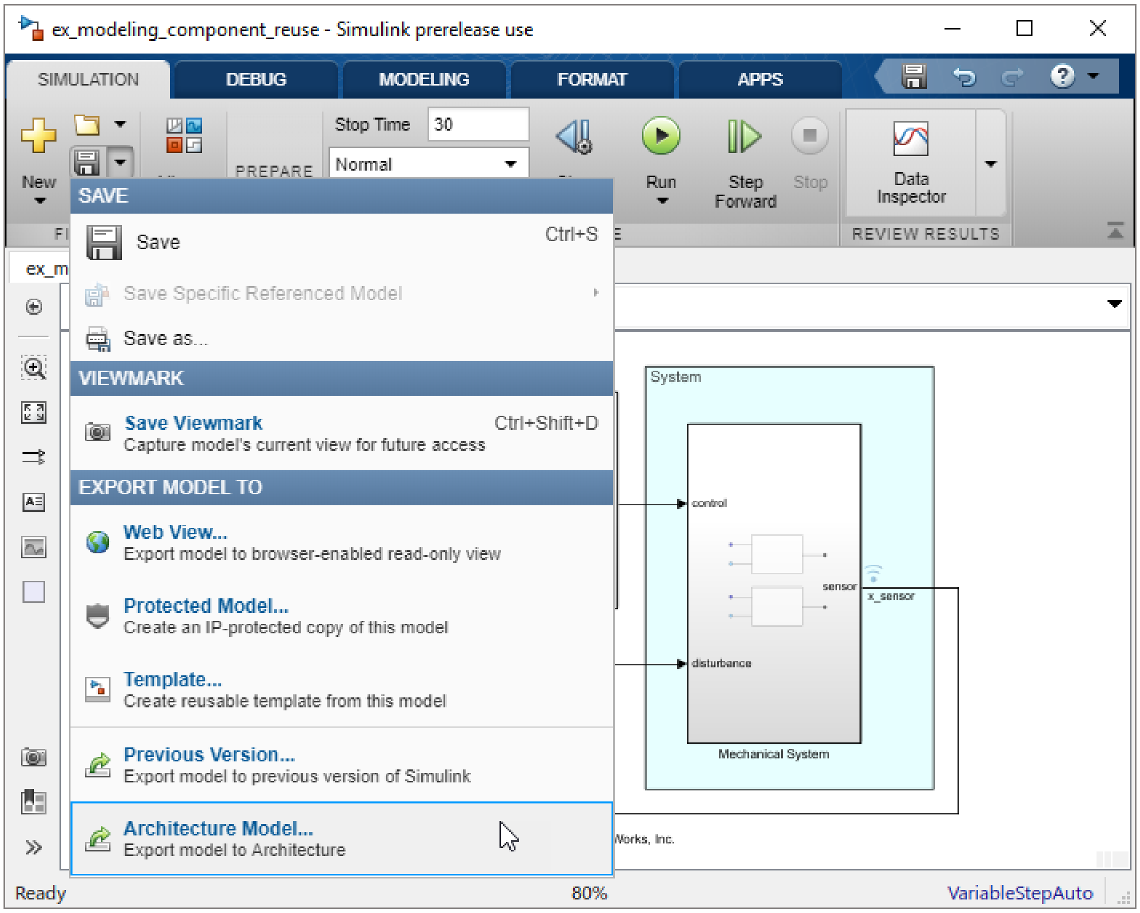
Extract Architecture from Simulink Model

You can use System Composer architecture editing and analysis capabilities on Simulink models. To do so, extract the architecture from a Simulink model. Model and Subsystem blocks, as well as all ports in a Simulink model represent architectural constructs, while all other blocks represent some kind of dynamic or algorithmic behavior. In the architecture model that you obtain from a Simulink model, you can choose to represent architectural constructs or link to behavior models.

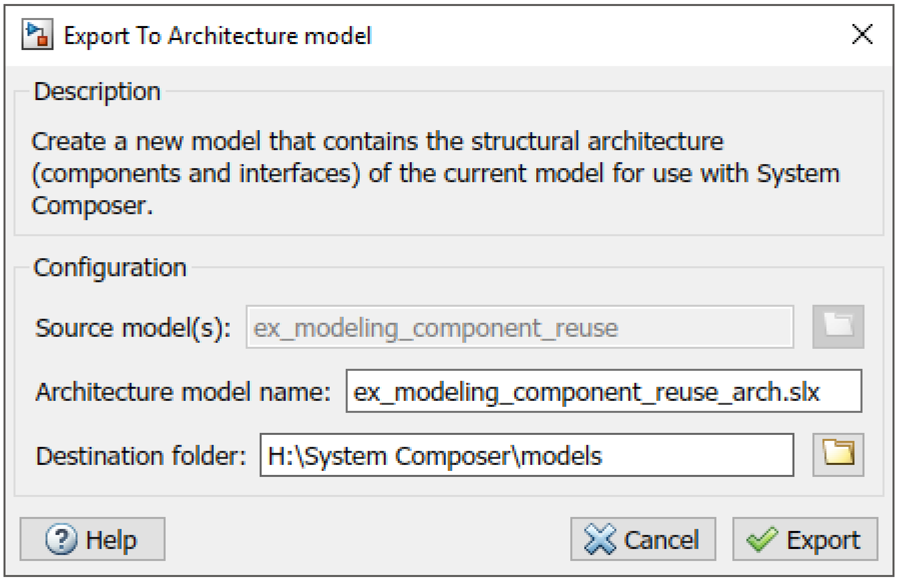
1 Open an example model.

```
openExample('ReferenceFilesForCollaborationExample')
```

2 On the **Simulation** tab, click the **Save** arrow. From the **Export Model To** list, select **Architecture Model**.



3 Provide a name and path for the architecture model.



Export To Architecture model

Description

Create a new model that contains the structural architecture (components and interfaces) of the current model for use with System Composer.

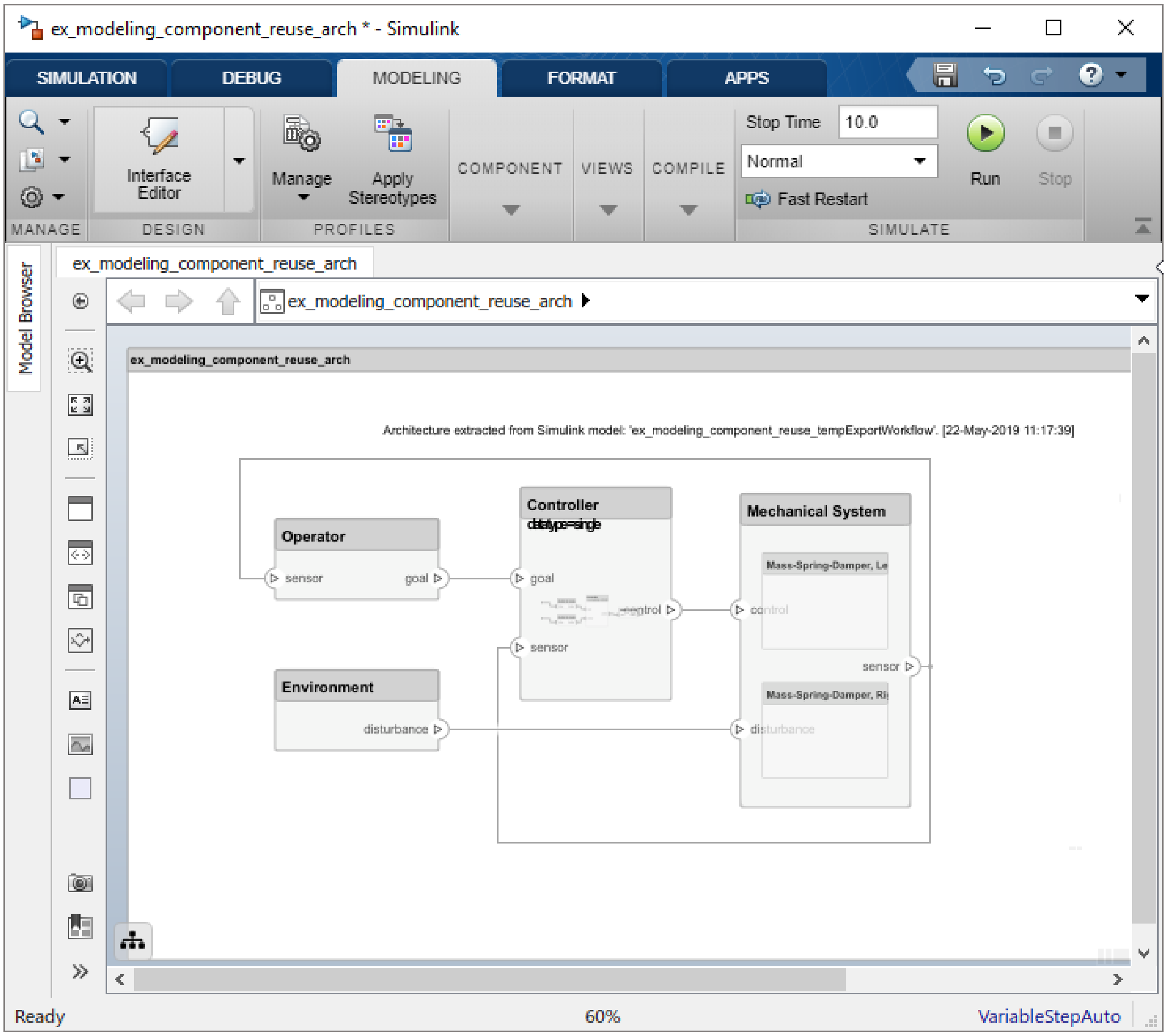
Configuration

Source model(s): ex_modeling_component_reuse

Architecture model name: ex_modeling_component_reuse_arch.slx

Destination folder: H:\System Composer\models

4 Click **Export**. A System Composer Editor window opens with an architecture model corresponding to the Simulink model.

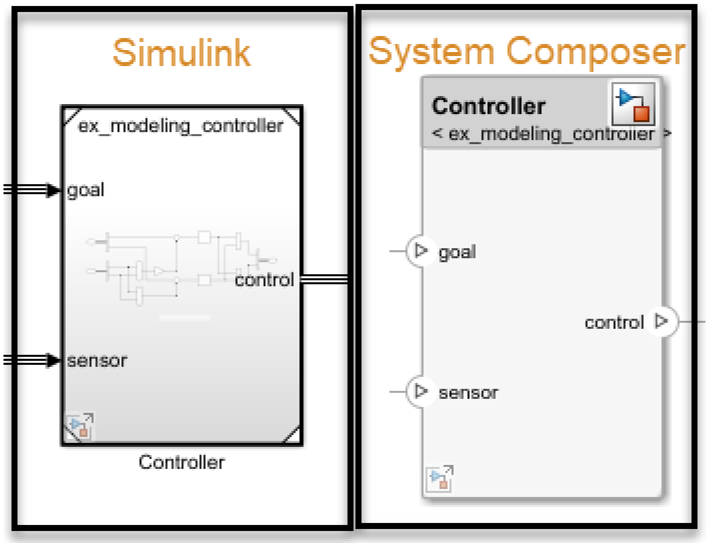


Architecture extracted from Simulink model: 'ex_modeling_component_reuse_tempExportWorkflow'. [22-May-2019 11:17:39]

Each subsystem in the Simulink model corresponds to a component in the architecture model so that the hierarchy in the architecture model reflects the hierarchy of the behavior model.

The requirements for subsystems and Model blocks in the Simulink model are preserved in the architecture model.

Any Model block in the Simulink model that references another model corresponds to a component that links to that same referenced model.



Buses at subsystem and Model block ports, as well as their dictionary links are preserved in the architecture model.

You can use the exported model to add architecture-related information such as interface definitions, nonfunctional properties for model elements and analyze the design.

See Also

`extractArchitectureFromSimulink`

More About

- “Extract the Architecture of a Simulink Model Using System Composer” on page 6-33
- “Implement Component Behavior in Simulink” on page 5-2
- “Add Stateflow Chart Behavior to Architecture Component” on page 5-7
- “Decompose and Reuse Components” on page 1-16

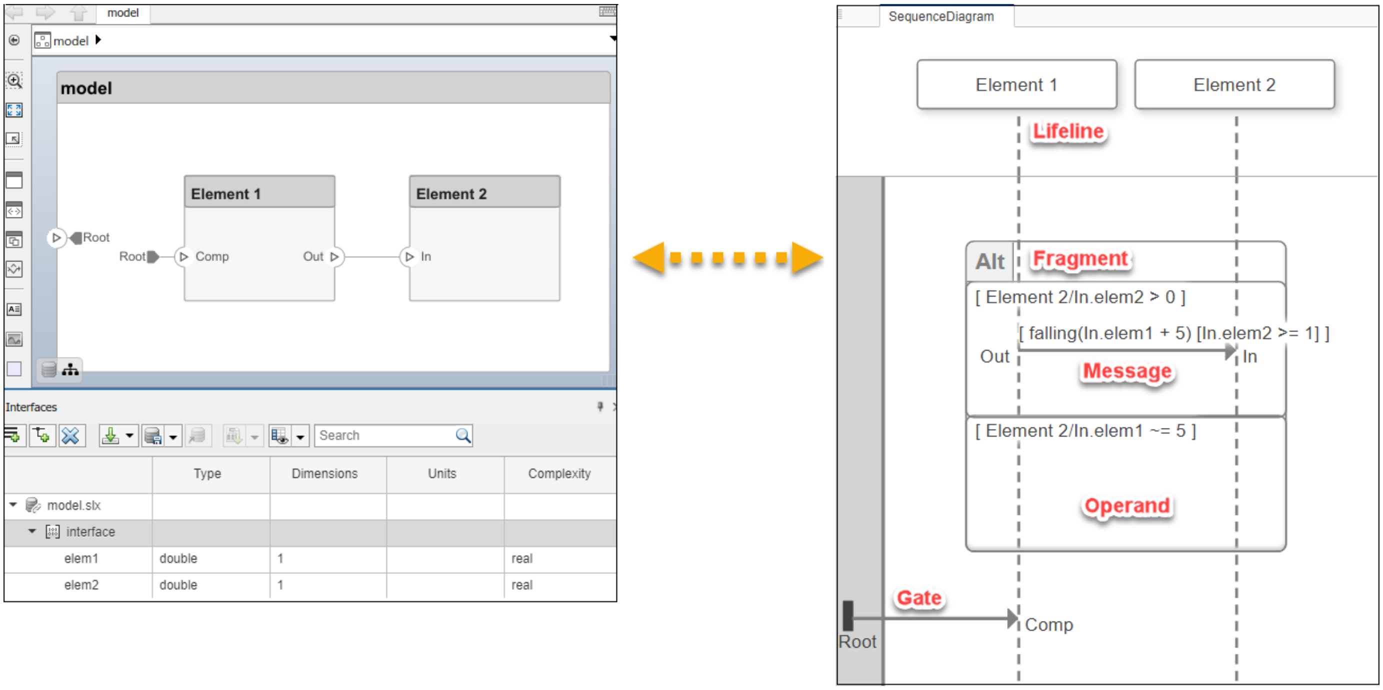
Define Sequence Diagrams

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.

You can use sequence diagrams in System Composer by accessing the Architecture Views Gallery. Sequence diagrams are integrated with architecture models. For more information on how to create and use sequence diagrams with an architectural model, see “Use Sequence Diagrams in the Views Gallery” on page 5-27.

In this example, you will learn about the basic terminology and functions of a sequence diagram in two stages.

- Add lifelines and messages with trigger conditions and constraint conditions to represent interactions.
- Include fragments and operands with constraint conditions to further specify the behavior of the interaction.

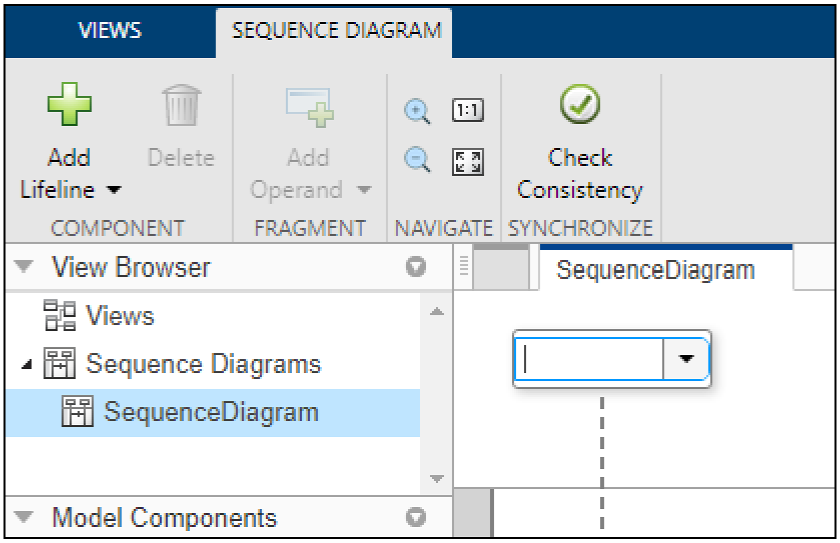


	Type	Dimensions	Units	Complexity
model.slx				
interface				
elem1	double	1		real
elem2	double	1		real

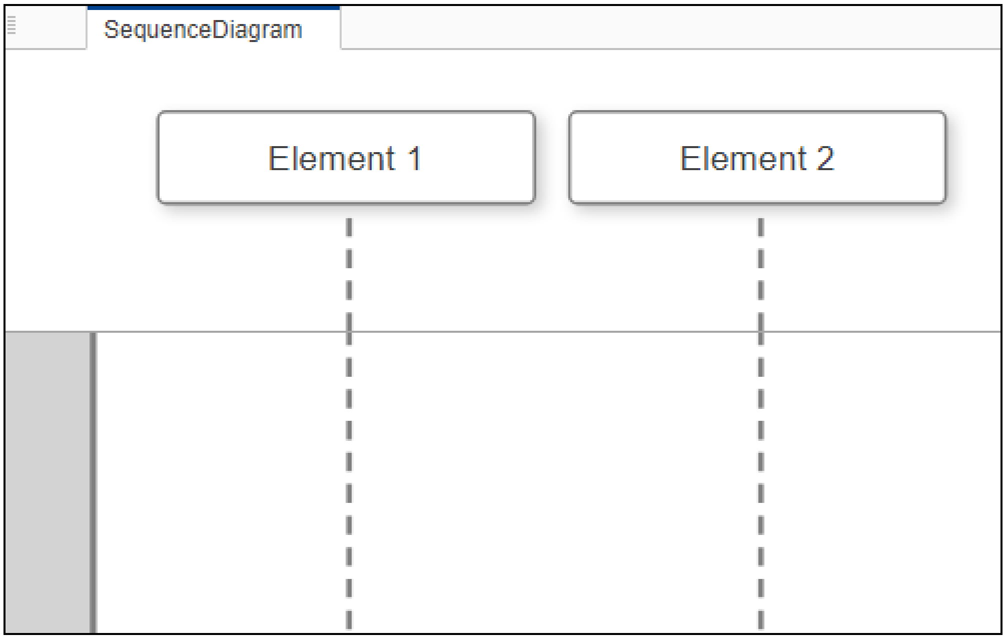
Add Lifelines and Messages

1 Create a new sequence diagram by navigating to **Views > Architecture Views**. The Architecture Views Gallery opens. Select **New Sequence Diagram** under the ✚ button to create a new sequence diagram.

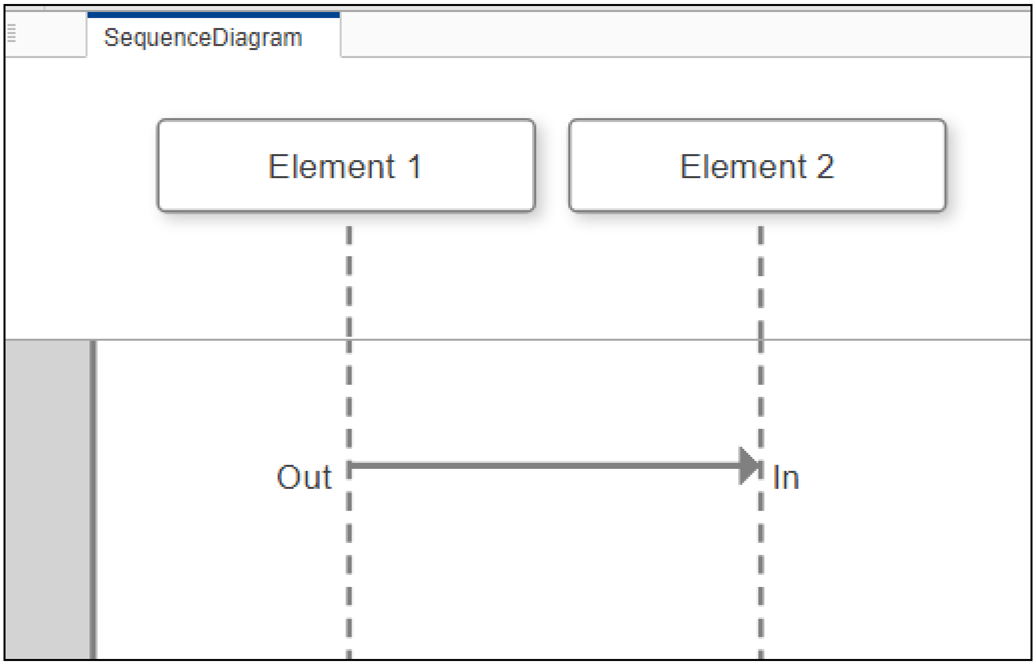
2 A new sequence diagram called `SequenceDiagram` is created in the View Browser and the Sequence Diagram tab becomes active. Select **Component > Add Lifeline** ✚ to add an element lifeline. A new lifeline with a vertical dashed line is created without a name.



3 Name the lifeline `Element 1` and create a second lifeline, `Element 2`.



4 Select the vertical dotted line for the `Element 1` lifeline. Click and drag to the `Element 2` lifeline. Specify the **To** and **From** message ends as `In` and `Out`, respectively.



5 Click on the message to see where to place the message condition. Enter a trigger condition with one of the following trigger events:

- `crossing`
- `rising`
- `falling`

For example, the message trigger condition could be specified as follows:

```
falling(In.elem1 + 5)
```

The signal name `In.elem1` must be a signal element in a signal interface associated with the port. For more information on interface management, see “Define Interfaces” on page 3-2.

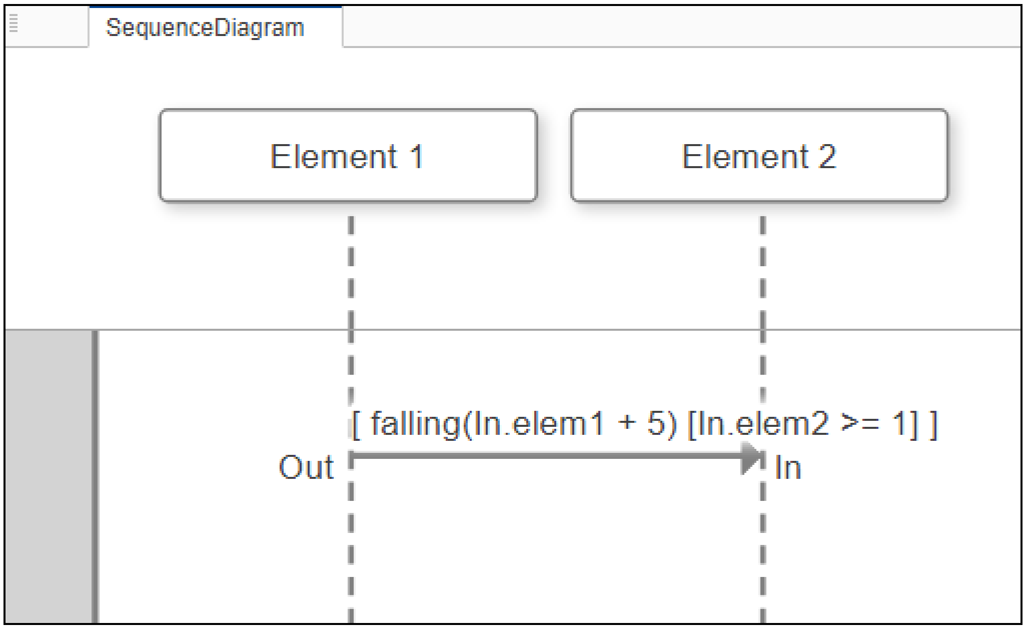
The trigger condition must be in this form:

```
triggerEvent(signalName (+|-) positiveReal)
```

A message trigger condition activates on a zero-crossing event when the value of the port signal is zero, starting from the specified value.

6 Add a constraint condition using a MATLAB boolean expression in square brackets. Constraint conditions consist of a boolean expression acting on a signal name.

```
[In.elem2 >= 1]
```



The constraint condition is an additional check after the trigger condition.

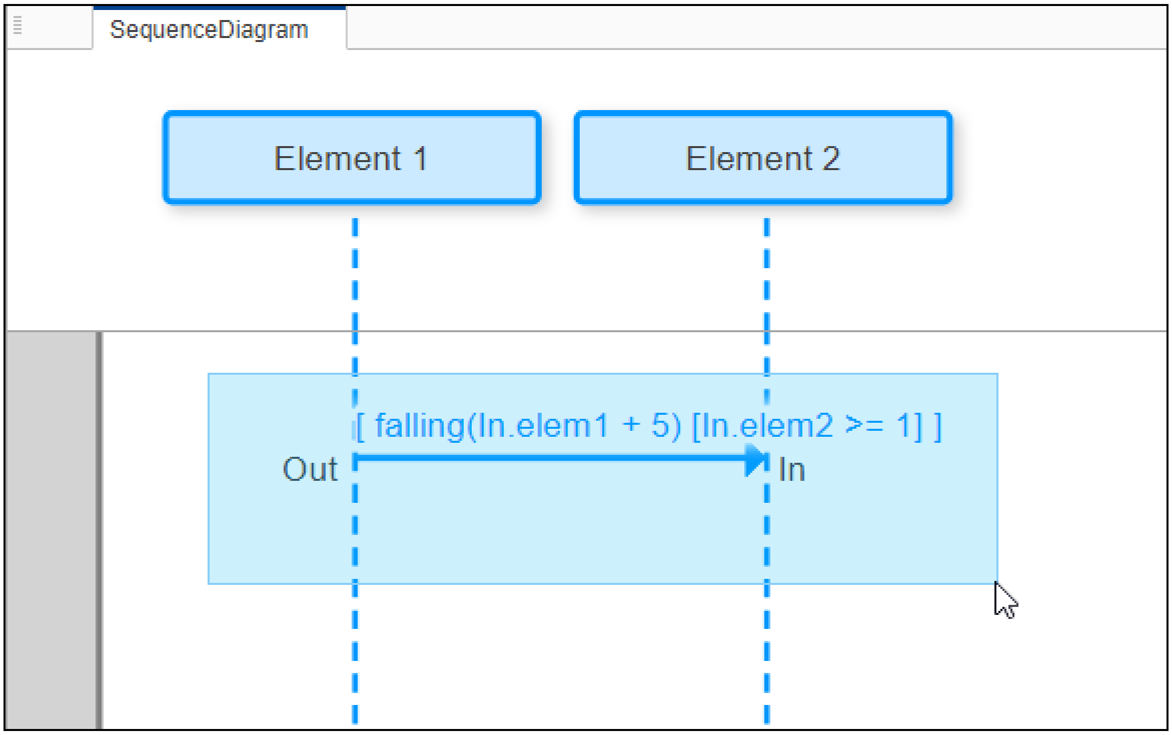
Note Only destination elements are supported for trigger conditions and constraints. In this example, `Out` is a source element and cannot be included.

Add Fragments and Operands

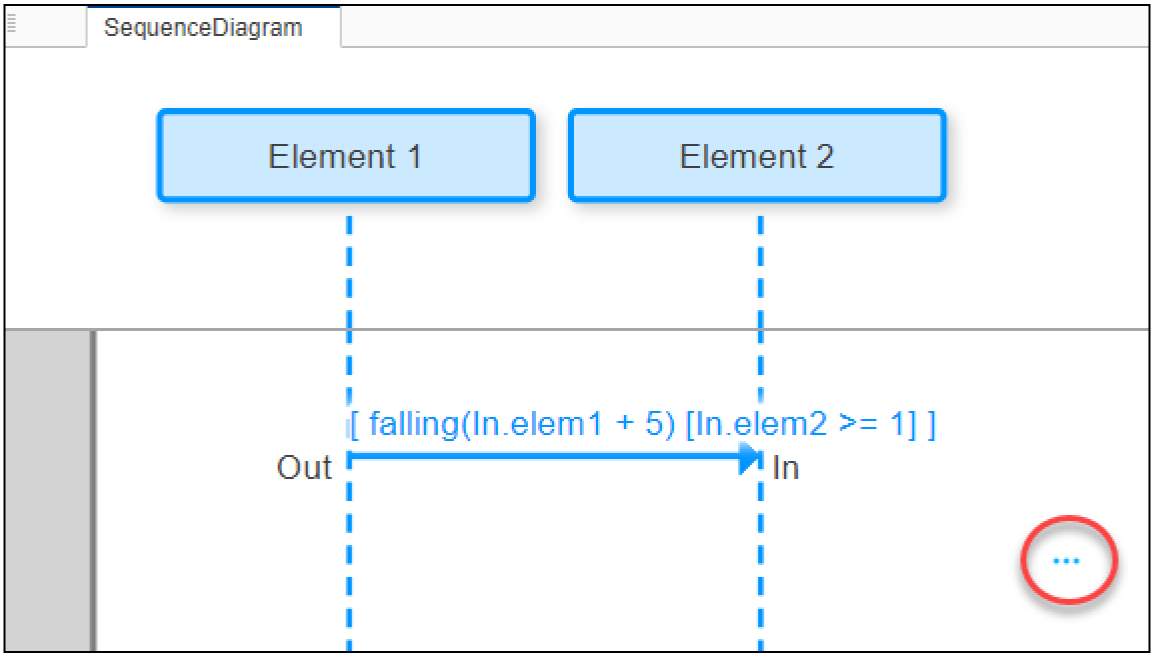
You can use composite fragments to enable control structures in sequence diagrams. Operands within composite fragments can be further specified with operand conditions composed of MATLAB boolean expressions between signal names.

To access a menu of fragments:

1. Click and drag to select the message.



2 Pause on the ellipsis (...) that appears to access the action bar.

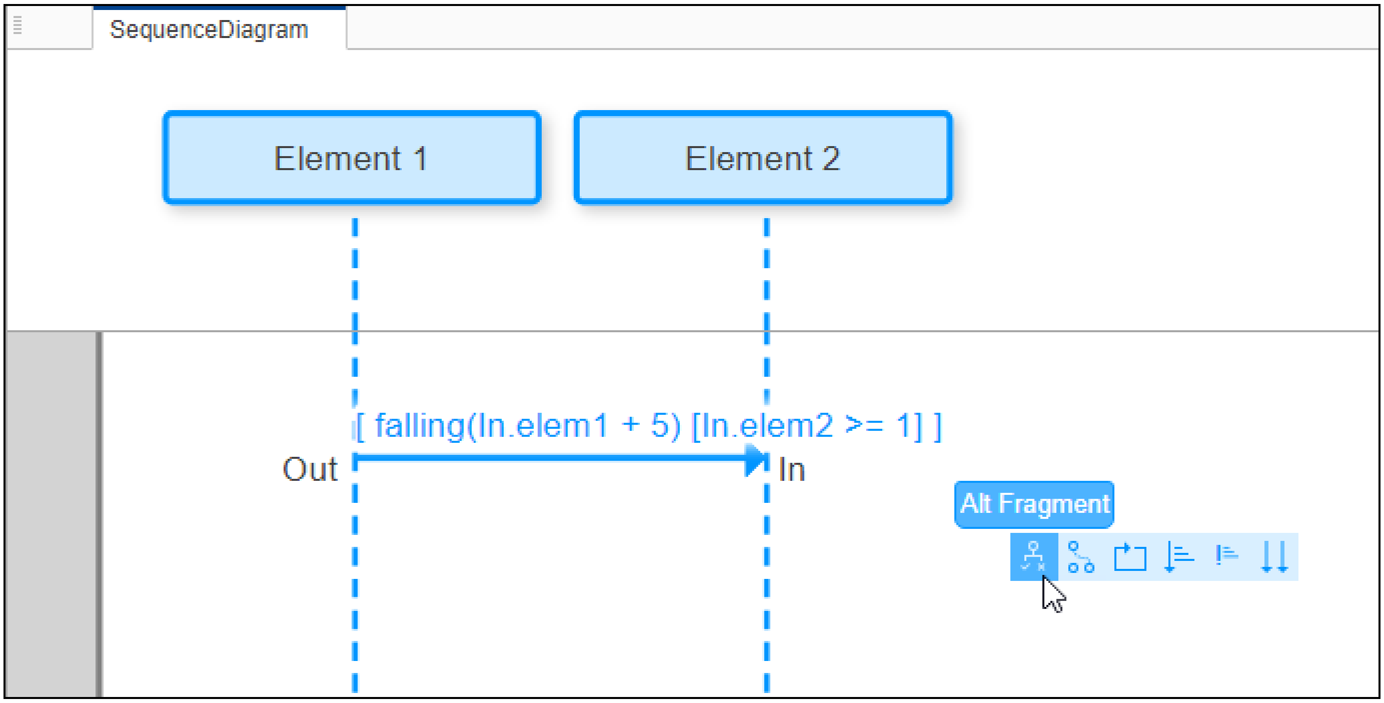


3 A list of composite fragments appears:

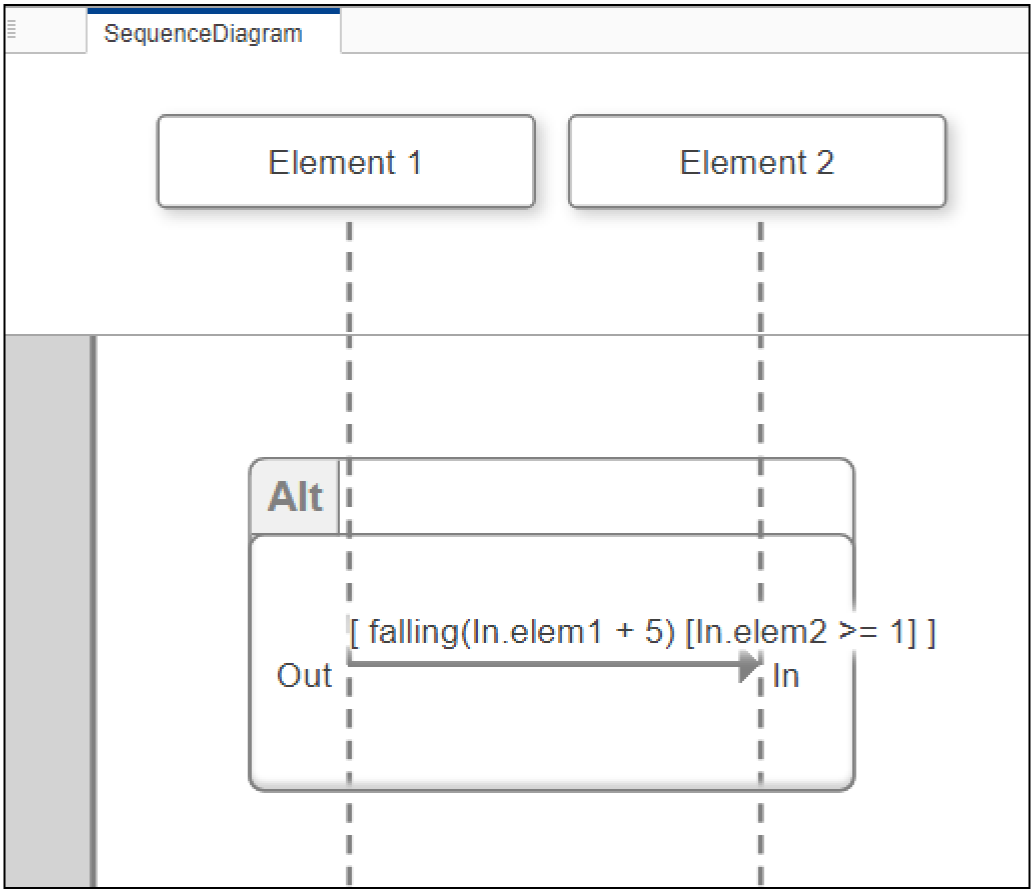
- `Alt Fragment`
- `Opt Fragment`
- `Loop Fragment`

- `Seq Fragment`
- `Strict Fragment`
- `Par Fragment`

Select `Alt Fragment`.



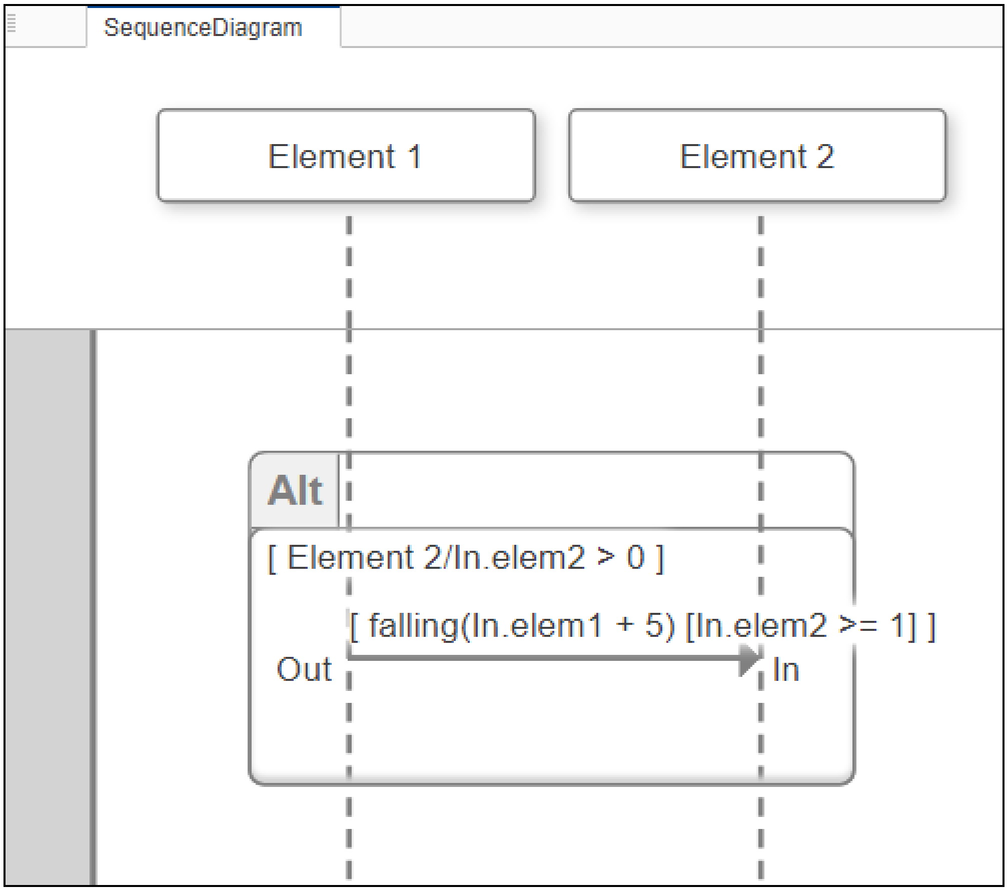
4 The `Alt Fragment` fragment is added to the sequence diagram message.



5 Select the composite fragment to enter an operand condition. Choose a fully qualified signal name and use a constraint condition relation.

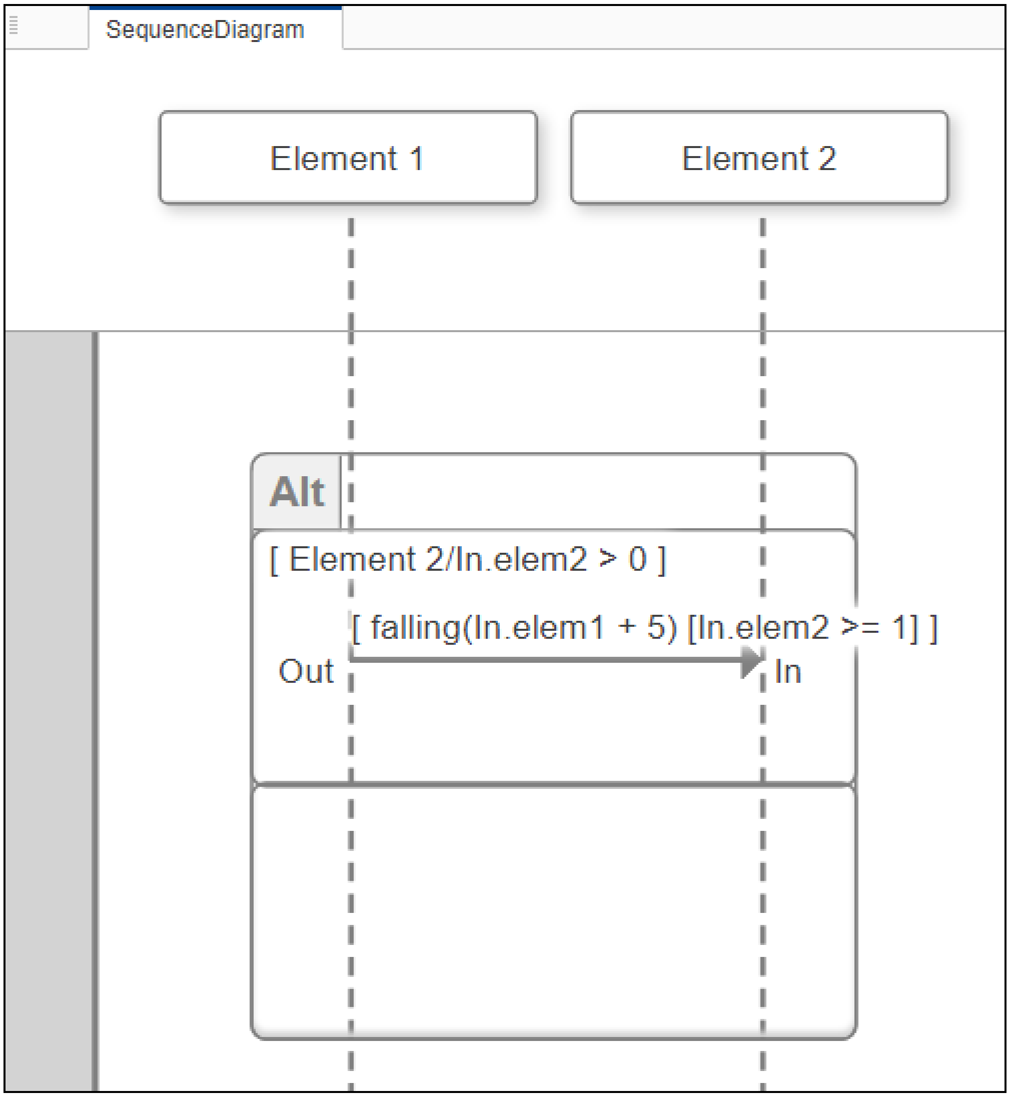
```
Element 2/In.elem2 > 0
```

The constraint determines when the alternative operand is accepted.


SequenceDiagram
Element 1
Element 2
Alt
[Element 2/In.elem2 > 0]
[falling(In.elem1 + 5) [In.elem2 >= 1]]
Out
In

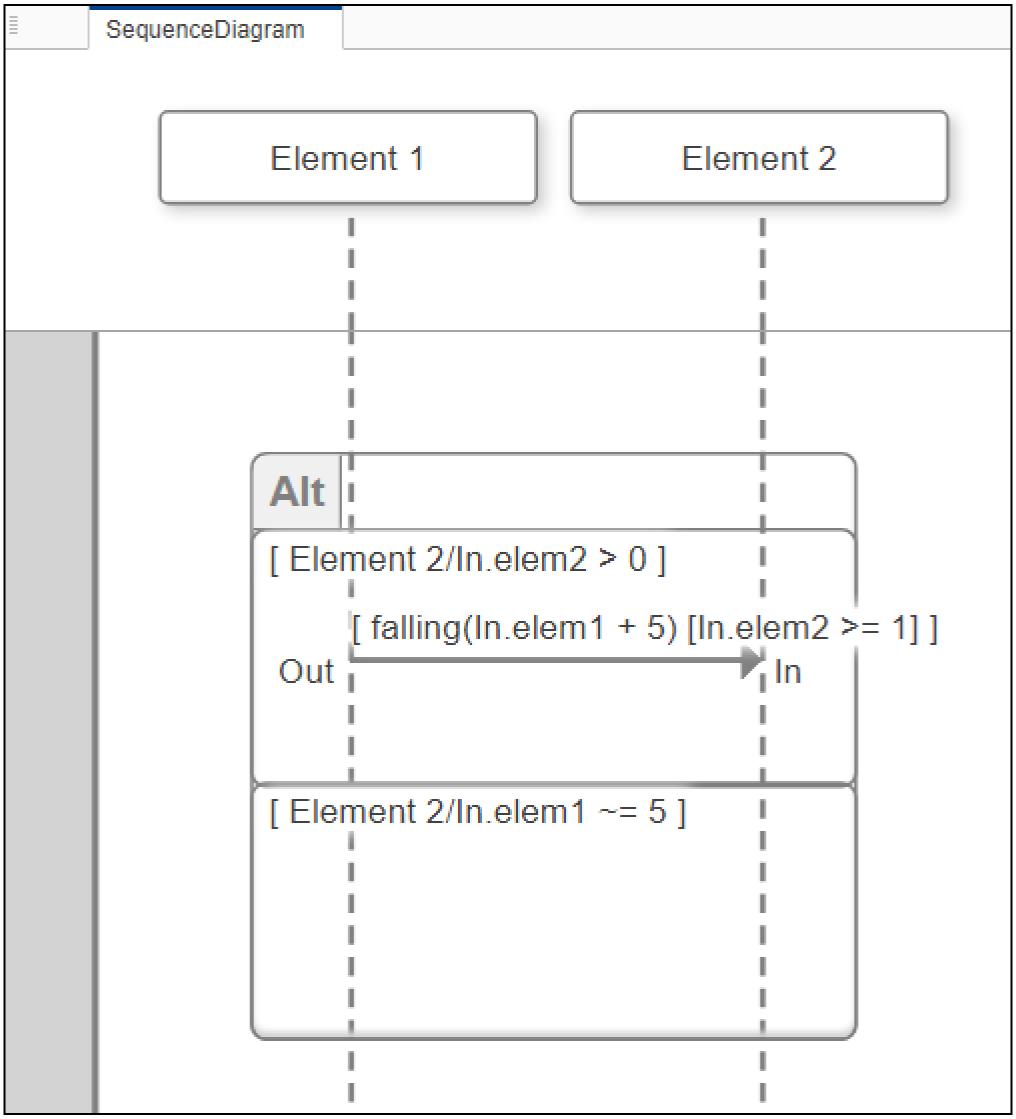
The message inside the operand can be executed only if the constraint condition is true.

6 Highlight the first operand under the `Alt Fragment` composite fragment and select **Fragment > Add Operand > Insert After**. A second operand is added.


SequenceDiagram
Element 1
Element 2
Alt
[Element 2/In.elem2 > 0]
[falling(In.elem1 + 5) [In.elem2 >= 1]]
Out
In

7 Add a constraint condition relation to the second operand.

```
Element 2/In.elem1 ~= 5
```


SequenceDiagram
Element 1
Element 2
Alt
[Element 2/In.elem2 > 0]
[falling(In.elem1 + 5) [In.elem2 >= 1]]
Out
In
[Element 2/In.elem1 ~= 5]

The second operand in an `Alt Fragment` fragment represents an `elseif` condition for which the message will not be executed.

View the Define Sequence Diagrams Example

You can view the final product of the workflow example for this topic.

Open the System Composer model that contains the sequence diagram.

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

Open the Architecture Views Gallery to view the sequence diagram.

```
openViews(model);
```

See Also

More About

- “Use Sequence Diagrams in the Views Gallery” on page 5-27
- “Implement Component Behavior in Simulink” on page 5-2
- “Add Stateflow Chart Behavior to Architecture Component” on page 5-7
- “Define Interfaces” on page 3-2

Use Sequence Diagrams in the Views Gallery

You can author sequence diagrams to describe expected system behavior as a sequence of interactions between components of a System Composer architecture model. New lifelines or messages authored on the sequence diagram are automatically reflected in the model. Incorporate model elements in a sequence diagram associated with the model the elements are from. You can create multiple sequence diagrams to represent different operational scenarios of the system.

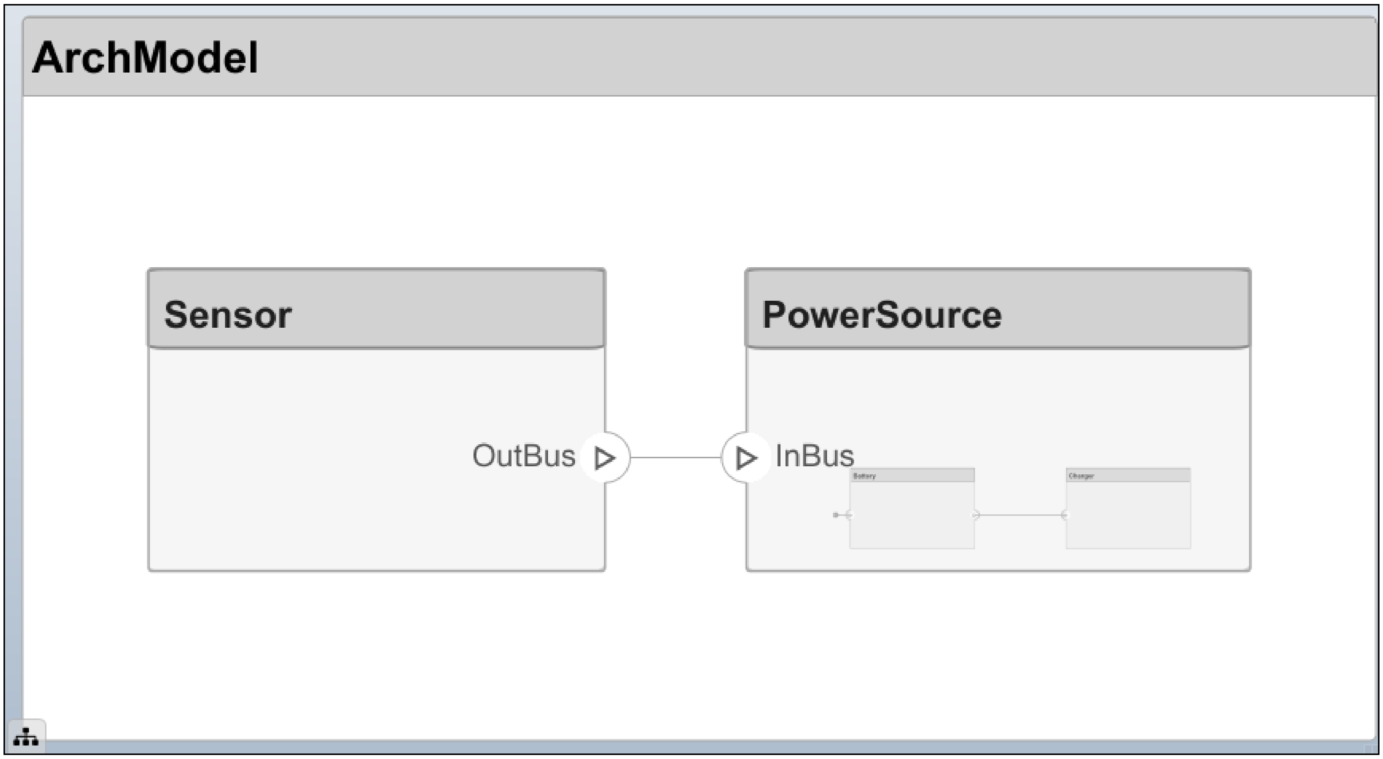
Sequence diagrams are integrated into the Architecture Views Gallery in System Composer. Lifelines in a sequence diagram correspond to components in an architecture model. Messages in a sequence diagram correspond to the connectors between components in an architecture model.

In this example, you will learn about using sequence diagrams in System Composer with emphasis on how to:

- Create a sequence diagram and co-create components and connections.
- Add child lifelines in a sequence diagram.
- Keep the architecture model and the sequence diagram in sync.

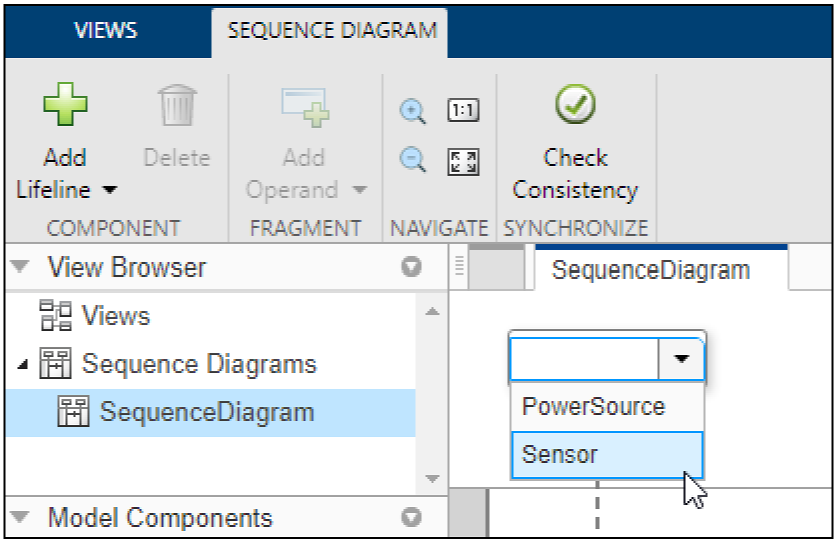
Create a Sequence Diagram

Create a basic architecture model in System Composer.



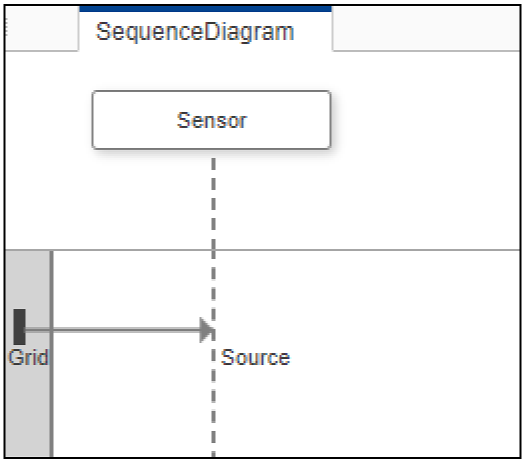
In the menu, navigate to **Views > Architecture Views** to open up the Architecture Views Gallery for your model. Select **New Sequence Diagram** under the ✚ button to create a new sequence diagram.

Select **Add Lifeline** ✚ from the menu. A box with a vertical dotted line appears on the canvas. This is the new lifeline. Click the down arrow on the lifeline to view available options. Select the component named Sensor to be represented by the lifeline.

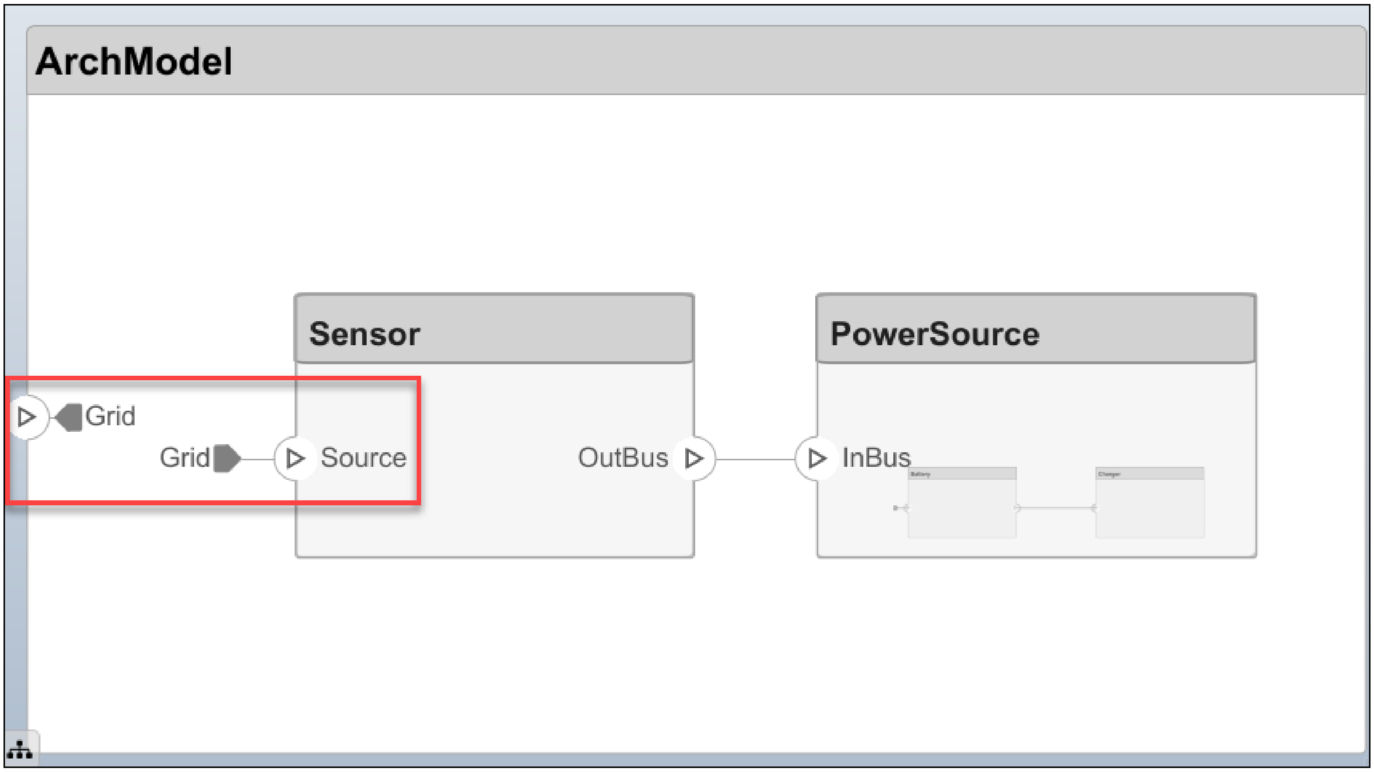


Create Sequence Diagram Gates

Select the gutter region, click, and drag to the lifeline. Name the **To** port Source and the **From** port Grid. See that a gate called Grid has been created with a message ending on the Sensor lifeline at the port Source.



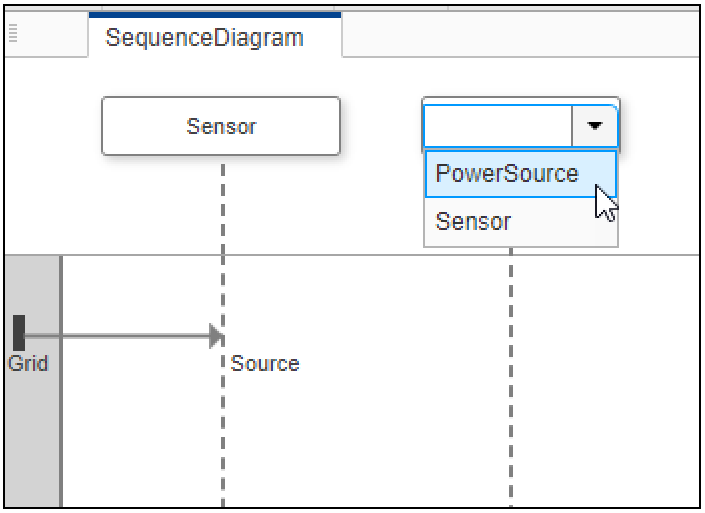
Return to the architecture diagram. Observe that Grid is a root architecture port connected to the Sensor component.



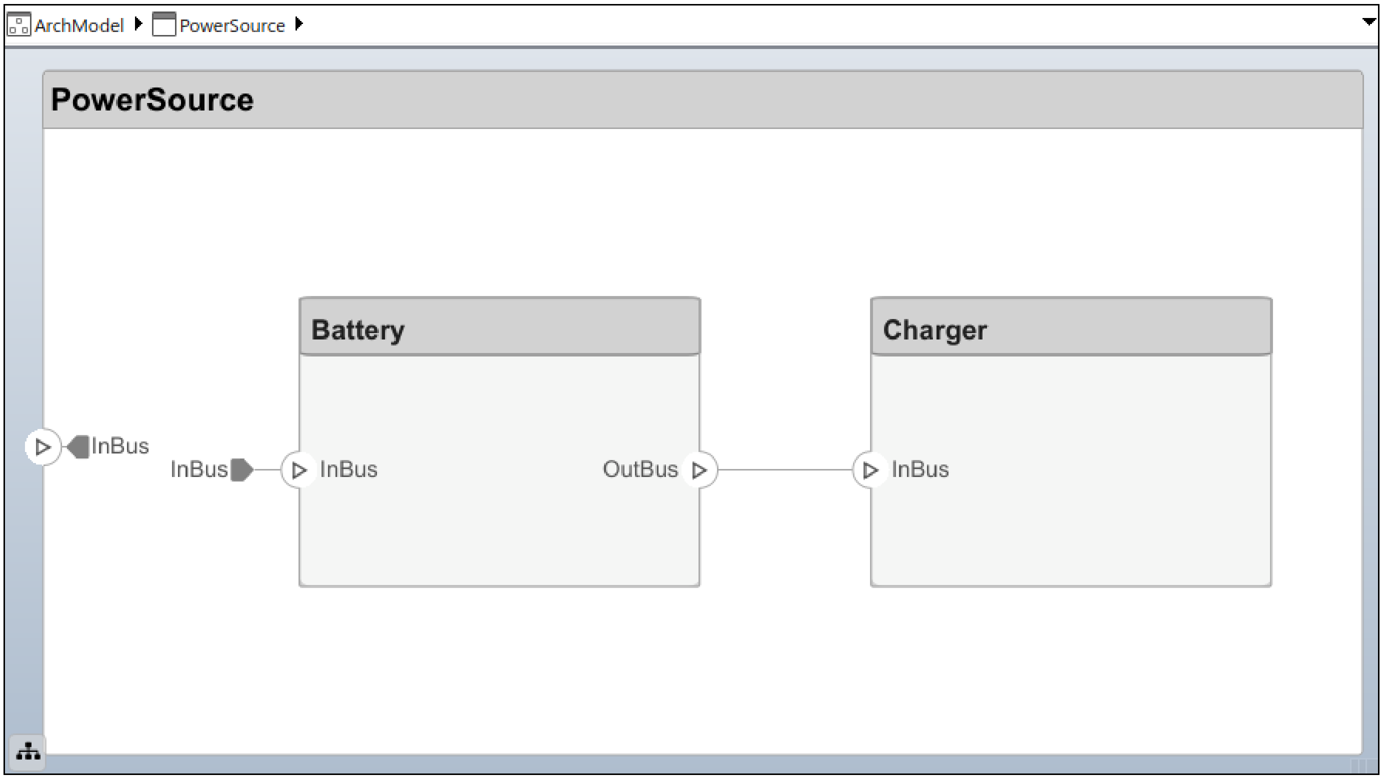
Add Child Lifelines in a Sequence Diagram

You can add child lifelines to a sequence diagram to represent model hierarchy and describe the interactions between them.

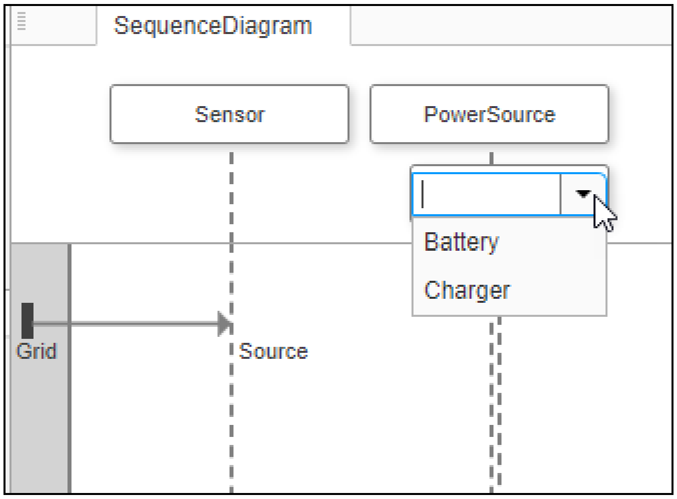
Select **Component > Add Lifeline** ✚ from the toolstrip menu. From the list that appears, select the PowerSource component.



Child components called Battery and Charger are located inside the PowerSource component.



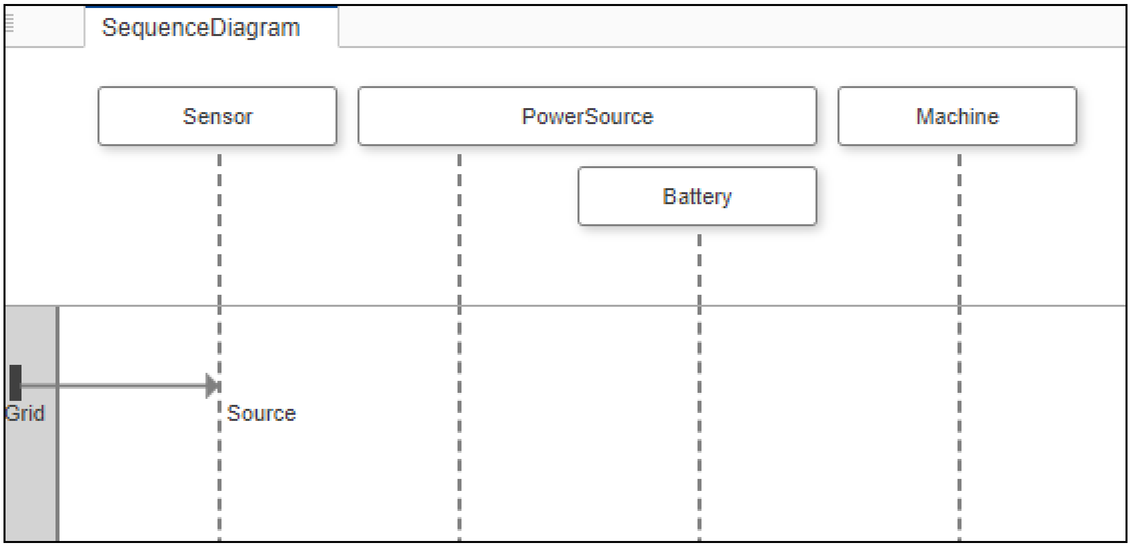
Select the `PowerSource` lifeline. Click the down arrow below **Component > Add Lifeline**, then select **Add Child Lifeline** Select `Battery`. The `Battery` child lifeline is now situated below `PowerSource` in the hierarchy.



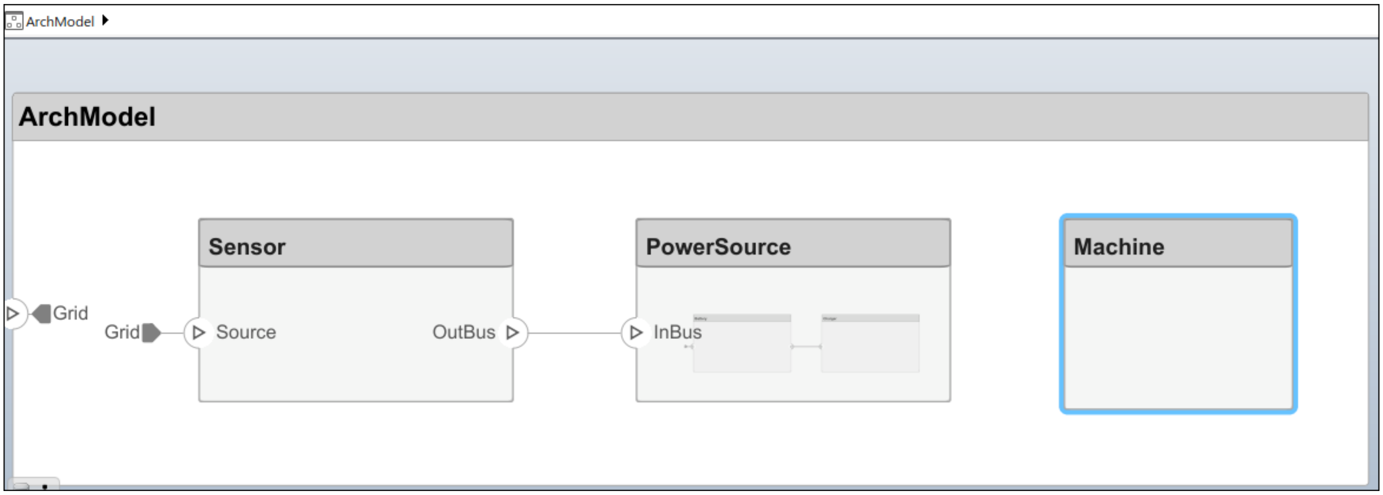
Co-Create Components

The co-creation workflow between the sequence diagram and the architecture model keeps the model synchronized as you make changes to the sequence diagram. Adding both lifelines and messages in a sequence diagram results in updates to the architecture model. This example shows component co-creation.

Select **Component > Add Lifeline** from the menu. Another box with a vertical dotted line appears on the canvas. In the lifeline box, enter the name of a new component named `Machine`.

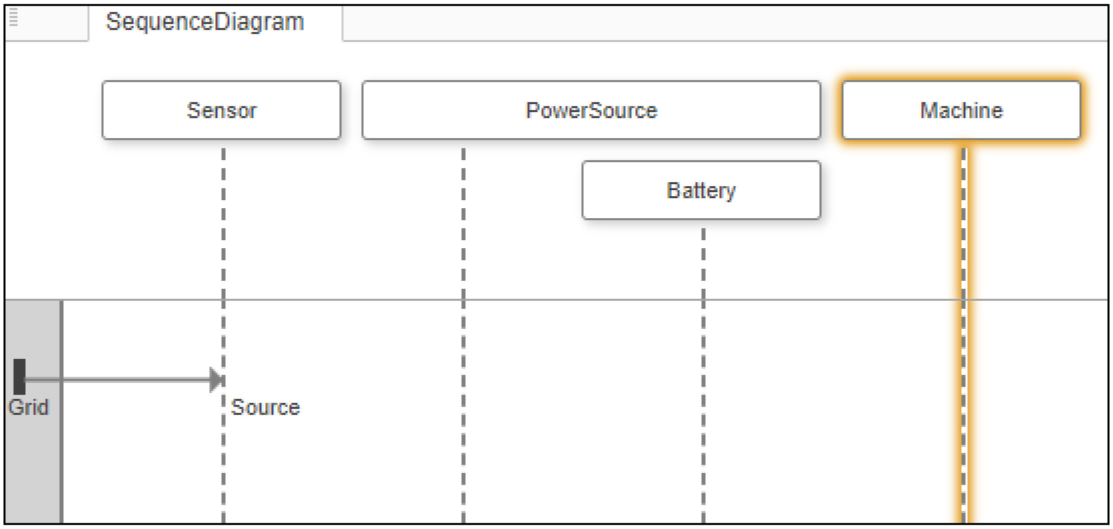


Observe that the `Machine` component is co-created in the architecture diagram.



Synchronize Between the Sequence Diagram and the Model

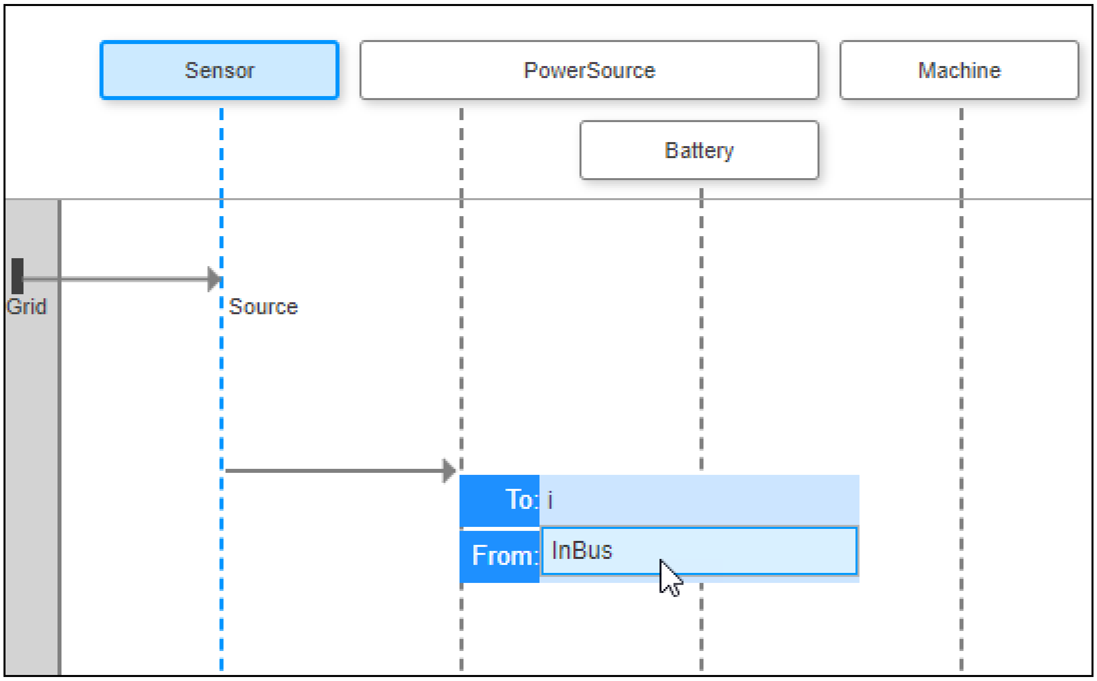
Remove the `Machine` component from the architecture diagram. Return to the sequence diagram and select **Synchronize > Check Consistency**. See that the `Machine` lifeline is highlighted, as it has no corresponding architectural component.



To restore the `Machine` component, either remove the `Machine` lifeline or select the undo button in the architecture model. Click **Check Consistency** again.

Create Messages in the Sequence Diagram

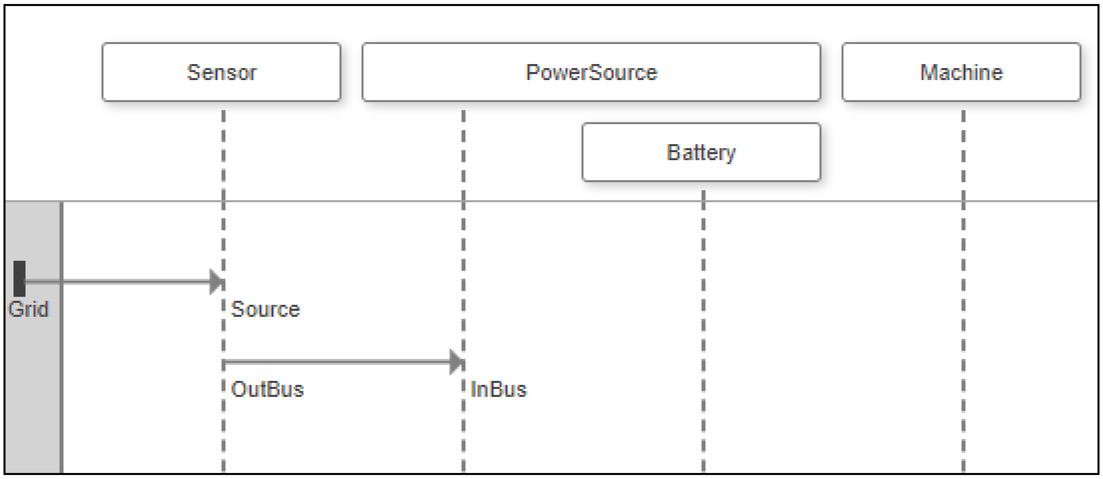
You can create a message from an existing connection. Draw a line from the `Sensor` lifeline to the `PowerSource` lifeline. Start to type `InBus`, which will automatically fill in as you type. When it does, select `InBus`.



To: i

From: InBus

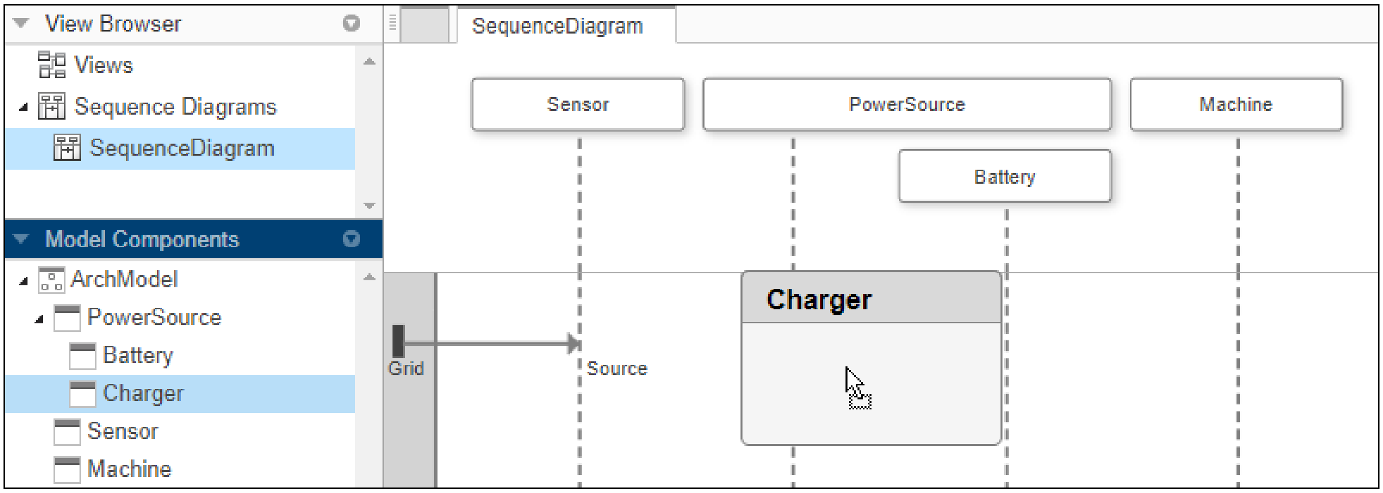
The message is created in the sequence diagram.



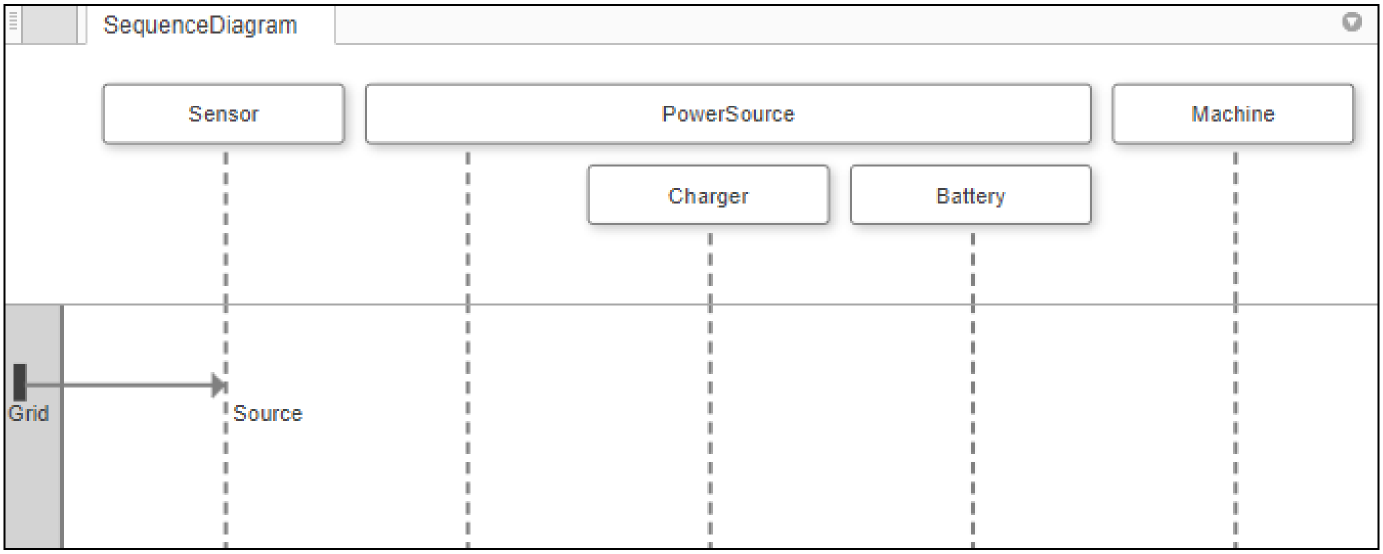
For more information on using message conditions, fragments, operands, and operand conditions in a sequence diagram, see “Define Sequence Diagrams” on page 5-16.

Click and Drag from the Model Browser

The Views Gallery model browser located on the bottom left of the canvas, is called Model Components. Click and drag the `Charger` child component into the sequence diagram.



The sequence diagram is updated with a new component.



Use Sequence Diagrams in the Views Gallery Example

You can view the final product of the workflow example for this topic.

Open the System Composer model that contains the sequence diagram.

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

Open the Architecture Views Gallery to view the sequence diagram.

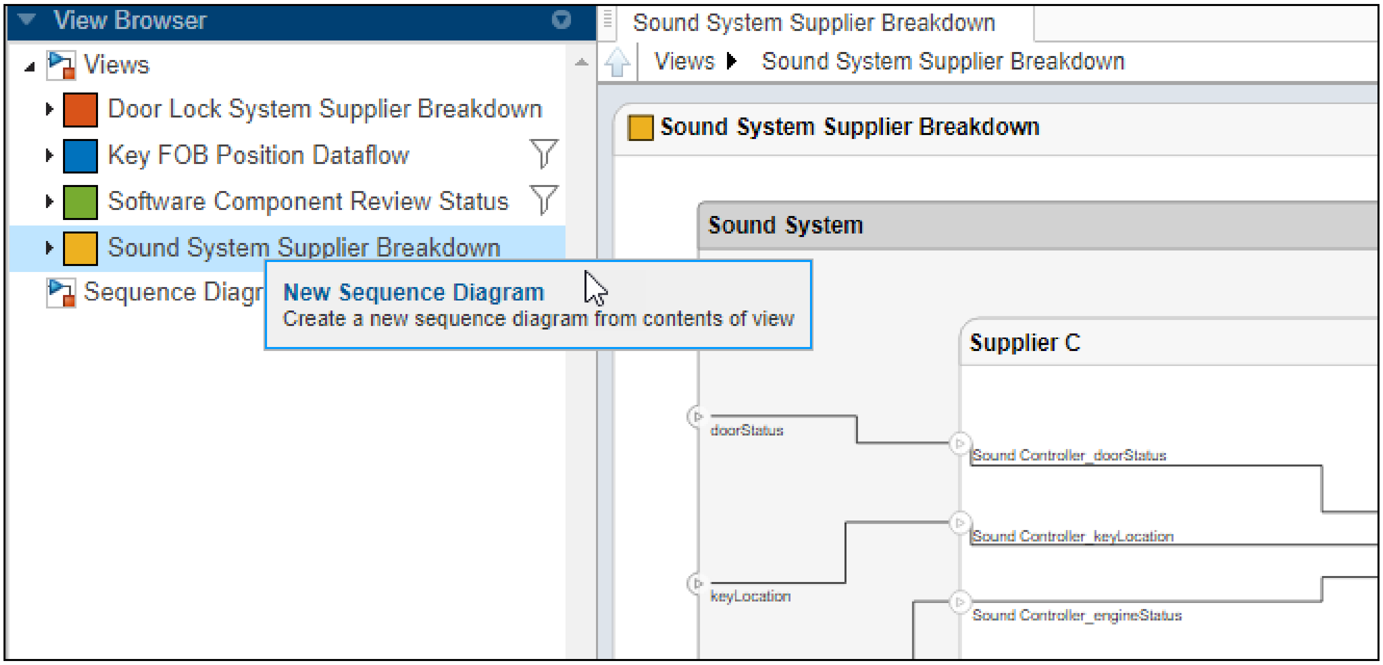
```
openViews(model);
```

Create a Sequence Diagram from a View

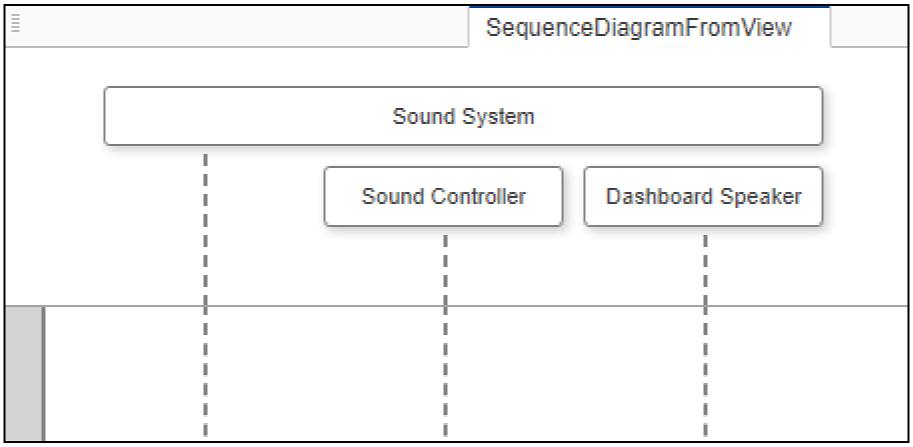
In the MATLAB Command Window, enter `scKeylessEntrySystem`. The architecture model opens in the Simulink Editor.

In the menu, navigate to **Views > Architecture Views** to open the Architecture Views Gallery for the model.

Right-click the `Sound System Supplier Breakdown` view and select **New Sequence Diagram**.


View Browser
Views
Door Lock System Supplier Breakdown
Key FOB Position Dataflow
Software Component Review Status
Sound System Supplier Breakdown
Sequence Diagr
New Sequence Diagram
Create a new sequence diagram from contents of view
Sound System Supplier Breakdown
Views
Sound System Supplier Breakdown
Sound System Supplier Breakdown
Sound System
Supplier C
doorStatus
keyLocation
Sound Controller_doorStatus
Sound Controller_keyLocation
Sound Controller_engineStatus

A new sequence diagram of lifelines is created with all the components from the view.



See Also

More About

- “Define Sequence Diagrams” on page 5-16
- “Implement Component Behavior in Simulink” on page 5-2
- “Add Stateflow Chart Behavior to Architecture Component” on page 5-7

Analyze Architecture Model

Create and Manage Allocations

This example shows how to create and manage System Composer™ allocations. Use allocations to establish a directed relationship from architecture elements (components, ports, and connectors) in one model to architecture elements in another model. One common use case for allocations is to establish relationships from software components to hardware components to indicate a deployment strategy.

This example uses the Tire Pressure Monitoring System (TPMS) project. To open the project, use this command:

```
scExampleTirePressureMonitorSystem
```

Create a New Allocation Set

You can create an allocation set using the Allocation Editor. An allocation set is a collection of allocation relationships between two models: a source model, and a target model. The allocation set is stored as an `.mldatx` file.

In this example, `TPMS_FunctionalArchitecture.slx` is the source model and the `TPMS_LogicalArchitecture.slx` is the target model.

To create an allocation set for these models, use this command.

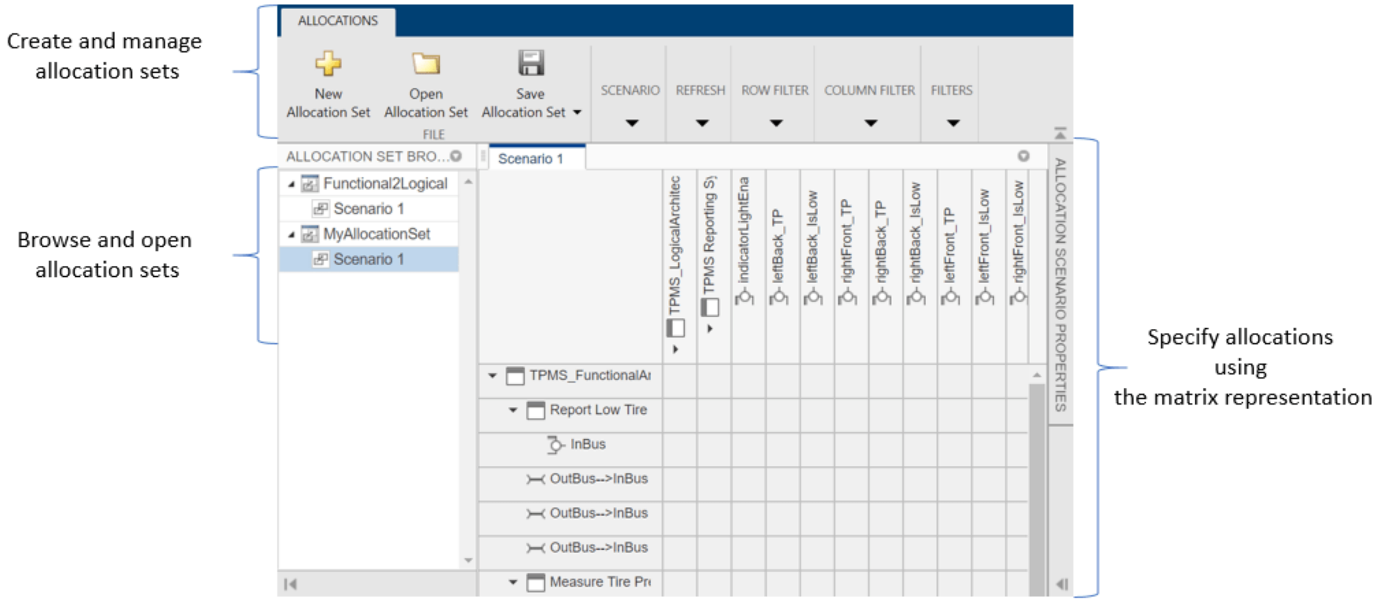
```
allocSet = systemcomposer.allocation.createAllocationSet(...
    'Functional2Logical', ...% Name of the allocation set
    'TPMS_FunctionalArchitecture', ... % Source model
    'TPMS_LogicalArchitecture' ... % Target model
     );
```

To see the allocation set, open the Allocation Editor by using the following command.

```
systemcomposer.allocation.editor;
```

The Allocation Editor has three parts: the toolstrip, the browser pane, and the allocation matrix.

- Use the toolstrip to create and manage allocation sets. For instance, you can use the **New Allocation Set** button to create a new allocation set between two models.
- Use the Allocation Set Browser pane to browse and open existing allocation sets.
- Use the allocation matrix to specify allocations between the source model elements in the first column and target model elements in the first row. You can create allocations programmatically or by double-clicking a cell in the matrix.



Create and manage allocation sets

Browse and open allocation sets

Specify allocations using the matrix representation

Create Allocations between Two Models

This example shows how to programmatically create allocations between two models in the TPMS project.

Get handles to the reporting functions in the functional architecture model.

```
functionalArch = systemcomposer.loadModel('TPMS_FunctionalArchitecture');
reportLevels = functionalArch.lookup('Path', 'TPMS_FunctionalArchitecture/Report Tire Pressure Le
reportLow = functionalArch.lookup('Path', 'TPMS_FunctionalArchitecture/Report Low Tire Pressure'
```

Get the handle to the TPMS reporting system component in the logical architecture model.

```
logicalArch = systemcomposer.loadModel('TPMS_LogicalArchitecture');
reportingSystem = logicalArch.lookup('Path', 'TPMS_LogicalArchitecture/TPMS Reporting System');
```

Create the allocations in the default scenario that is created.\

```
defaultScenario = allocSet.getScenario('Scenario 1');
defaultScenario.allocate(reportLevels, reportingSystem);
defaultScenario.allocate(reportLow, reportingSystem);
```

Save the allocation set.

```
allocSet.save;
```

Optionally, you can delete the allocation between reporting low tire pressure and the reporting system.

```
defaultScenario.deallocate(reportLow, reportingSystem);
```

See Also

allocate | editor | getScenario | systemcomposer.allocation.AllocationScenario | systemcomposer.allocation.AllocationSet

More About

- "Allocate Architectures in a Tire Pressure Monitoring System" on page 6-5
- "Simulating Mobile Robot with System Composer Workflow" on page 6-46

Allocate Architectures in a Tire Pressure Monitoring System

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

Overview

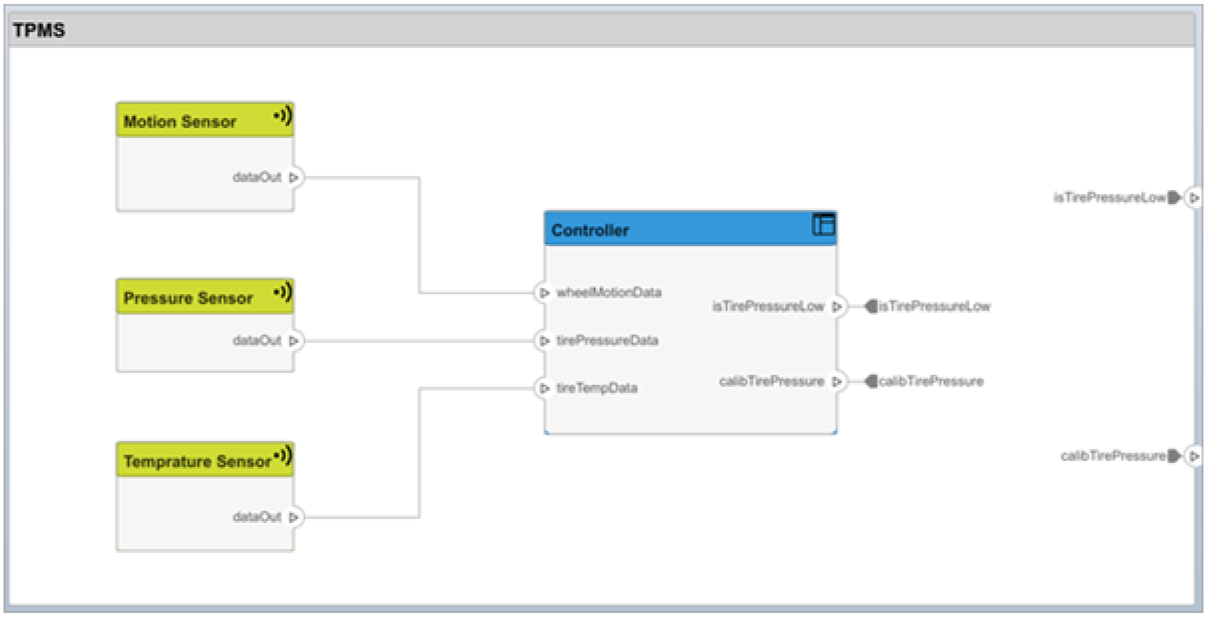
In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system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

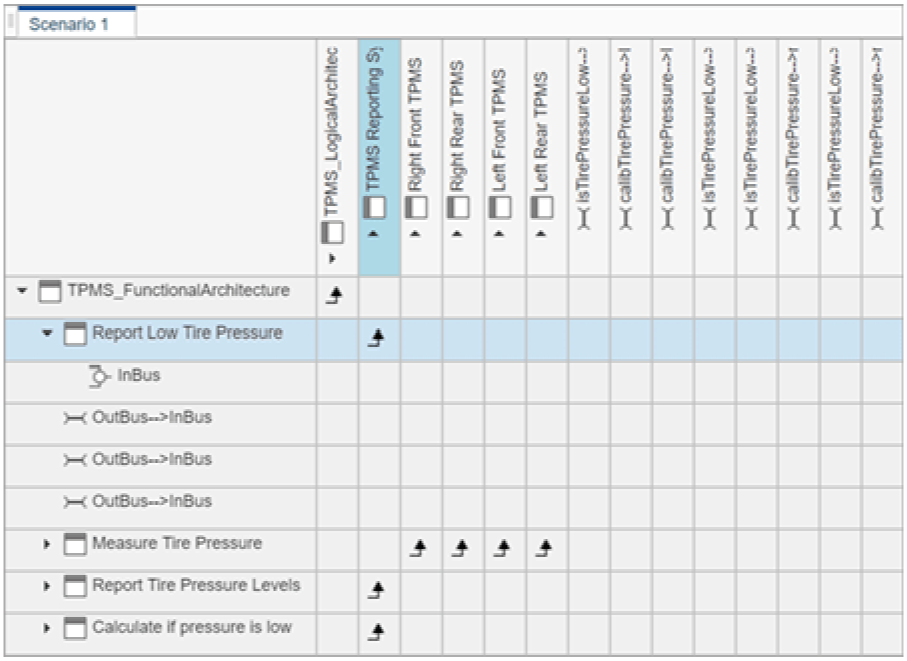
The allocation process is defined as linking these three architectures that fully describe the system. The linking captures the information about each architectural layer and makes it accessible to the others.

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.



Scenario 1

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

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
```

See Also

`getAllocatedFrom` | `getAllocatedTo` | `getScenario` | `load`

More About

- “Create and Manage Allocations” on page 6-2

Analyze Architecture

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.

Write static analyses based on element properties to perform data-driven trade studies and verify system requirements. Consider an electromechanical system where there is a trade-off between cost and weight, and lighter components tend to cost more. The decision process involves analyzing the overall cost and weight of the system based on the properties of its elements, and iterating on the properties to arrive at a solution that is acceptable both from the cost and weight perspective.

The analysis workflow consists of these steps:

1. Define a profile containing a set of stereotypes that describe some analyzable properties (for example, cost and weight).
2. Apply the profile to an architecture model and add stereotypes from that profile to elements of the model (components, ports, or connectors).
3. Specify values for the properties on those elements.
4. Create an instance of the architecture model, which is a tree of elements, corresponding to the model hierarchy with all shared architectures expanded and a variant configuration applied.
5. Write an analysis function to compute values necessary for the study. This is a static constraint solver for parametrics and values of related properties captured in the system model.
6. Run the analysis function.

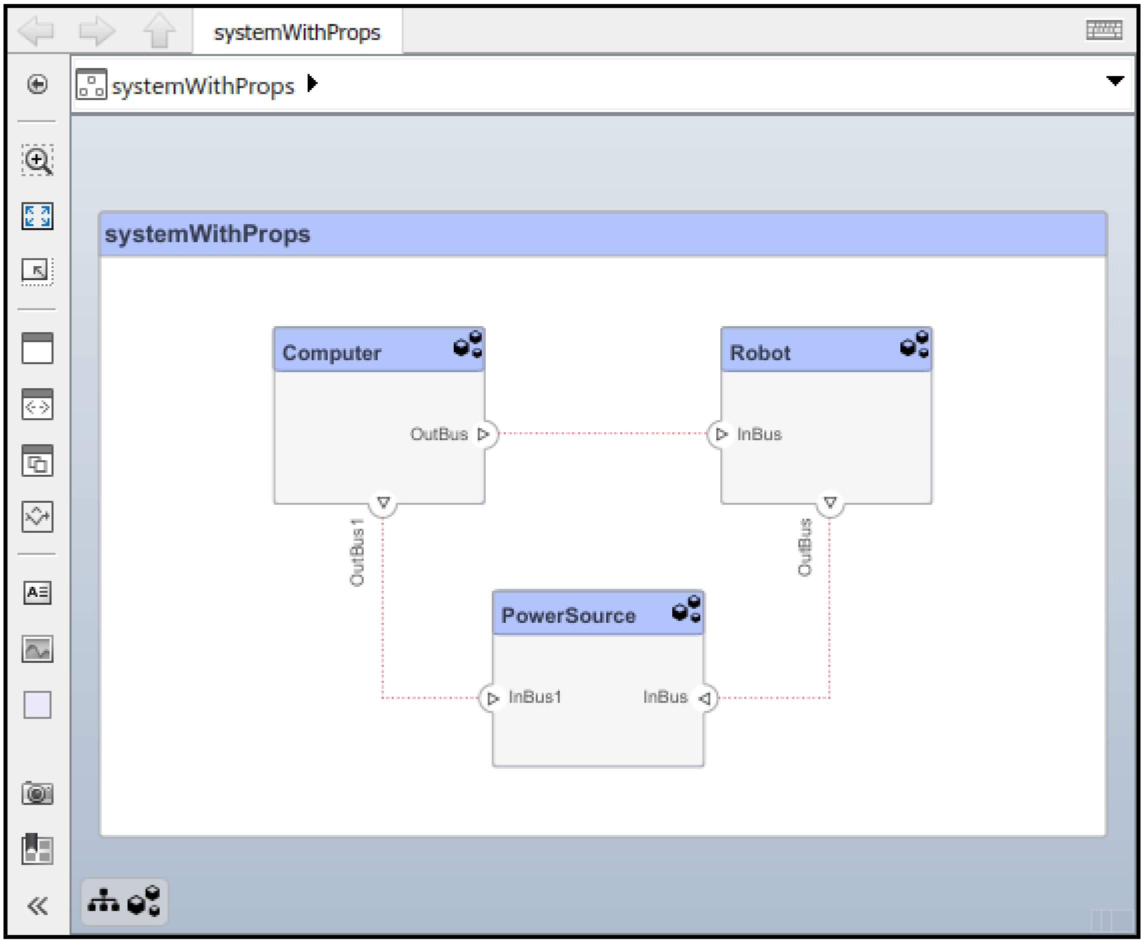
Set Properties for Analysis

This example shows how to enable analysis by adding stereotypes to model elements and setting property values. The model provides the basis to analyze the trade-off between total cost and weight of the components in a simple architecture model of a robot system.

Load the Model

Open the `systemWithProps` model.

```
systemWithProps
```



Import a Profile

Enable analysis of properties by first importing a profile. In the **Profiles** section of the toolstrip, click **Manage** > **Import** and browse to the profile.

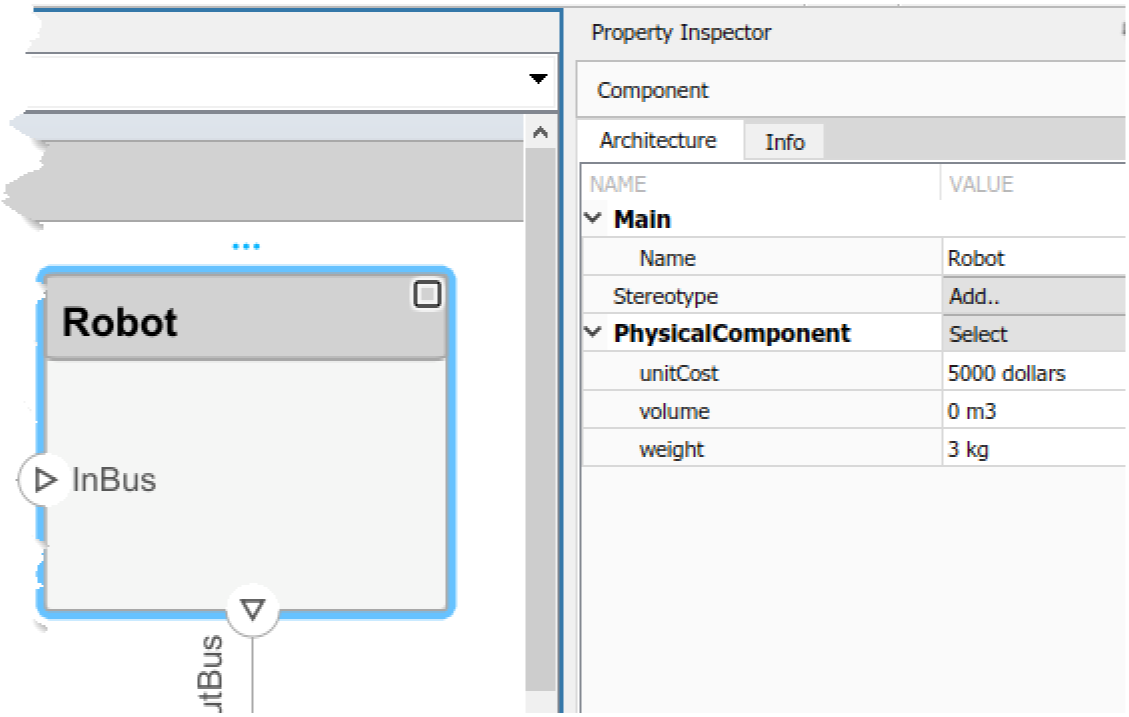
Apply Stereotypes to Model Elements

Apply stereotypes to all model elements that are part of the analysis. Use the menu items that apply stereotypes to all elements of a certain type. Select **Apply Stereotypes** > **Apply to** and then **Components** > **This layer**. Make sure you apply the stereotype to the top-level component, if a cumulative value is to be computed.

Set Property Values

Set property values for each model element.

1. Select the model element.
2. In the Property Inspector, expand the stereotype name and type values for properties.

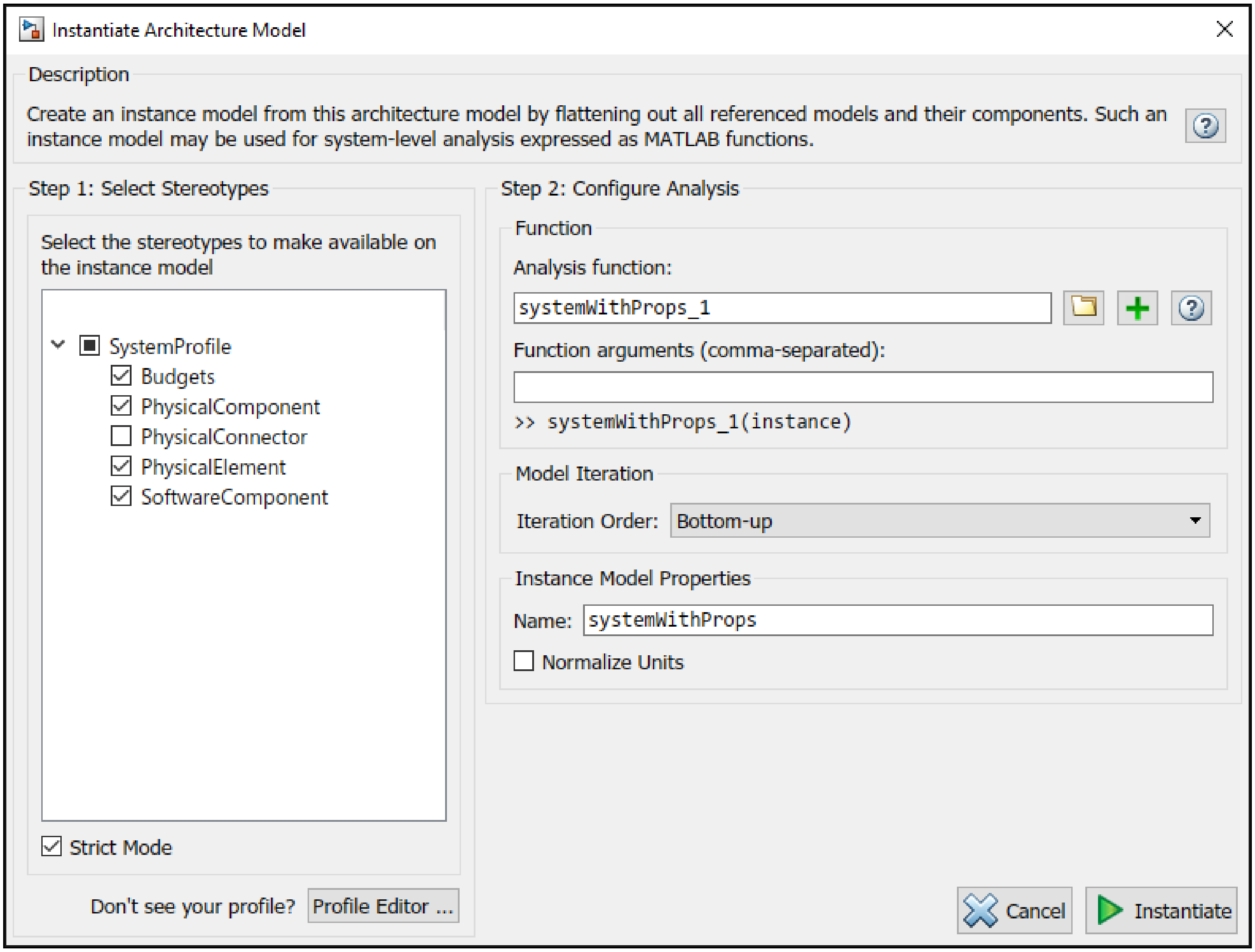


NAME	VALUE
Main	
Name	Robot
Stereotype	Add..
PhysicalComponent	Select
unitCost	5000 dollars
volume	0 m3
weight	3 kg

Create a Model Instance for Analysis

Create an instance of the architecture model that you can use for analysis. 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.

In the **Views** section, select **Analysis Model** > **Analysis Model**. In this dialog box, specify all the parameters required to create and view an analysis model.



Instantiate Architecture Model

Description

Create an instance model from this architecture model by flattening out all referenced models and their components. Such an instance model may be used for system-level analysis expressed as MATLAB functions.

Step 1: Select Stereotypes

Select the stereotypes to make available on the instance model

- ■ SystemProfile
  - ☑ Budgets
  - ☑ PhysicalComponent
  - □ PhysicalConnector
  - ☑ PhysicalElement
  - ☑ SoftwareComponent

☑ Strict Mode

Don't see your profile?

Step 2: Configure Analysis

Function

Analysis function: `systemWithProps_1`

Function arguments (comma-separated):

```
>> systemWithProps_1(instance)
```

Model Iteration

Iteration Order: Bottom-up

Instance Model Properties

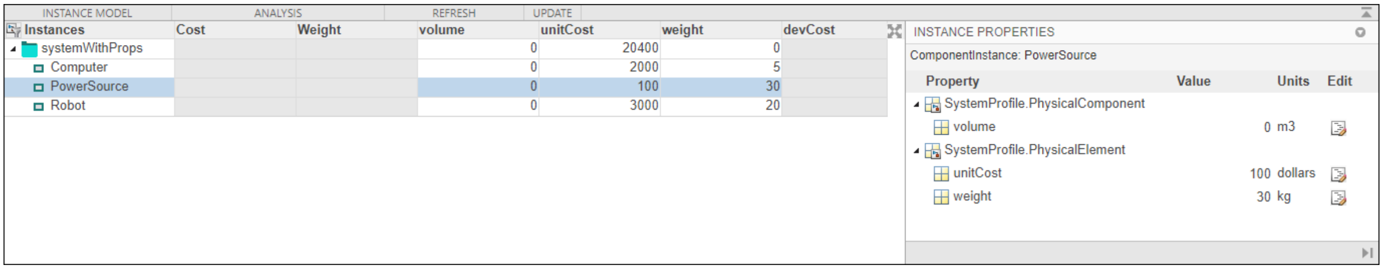
Name: `systemWithProps`

□ Normalize Units

The stereotypes tree lists the stereotypes of all profiles that have been loaded in the current session and allows you to select those whose properties should be available in the instance model. You can browse for an analysis function, create a new one, or skip analysis at this point. If the analysis function requires inputs other than elements in the model (such as an exchange rate to compute cost) enter it in **Function arguments**. Select a mode for iterating through model elements, for example, `Bottom-up` to move from the leaves of the tree to the root.

Note Strict Mode ensures instances only get properties if the instance's specification has the stereotype applied.

To view the instance, click **Instantiate**.



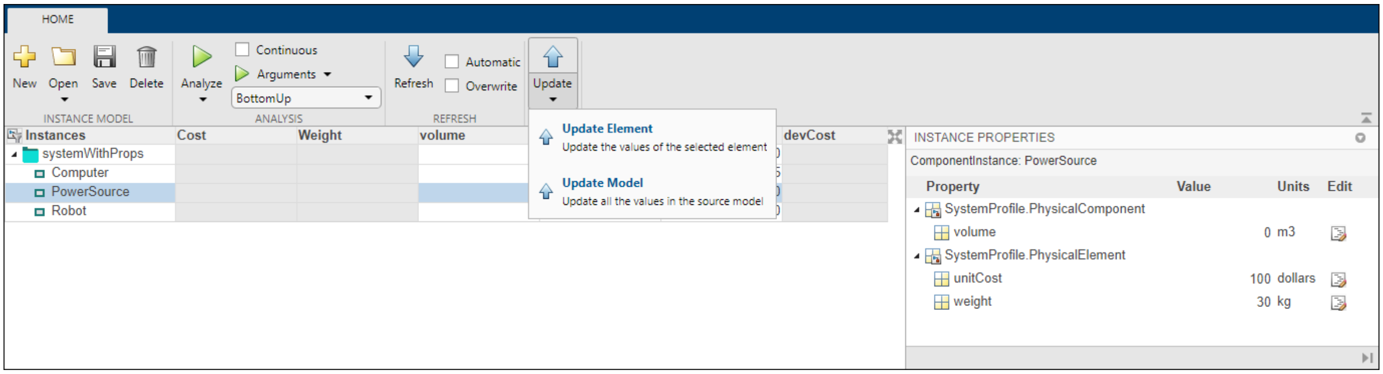
Instances	Cost	Weight	volume	unitCost	weight	devCost
systemWithProps			0	20400	0	
Computer			0	2000	5	
PowerSource			0	100	30	
Robot			0	3000	20	

INSTANCE PROPERTIES

ComponentInstance: PowerSource

Property	Value	Units	Edit
SystemProfile.PhysicalComponent			
volume	0	m3	
SystemProfile.PhysicalElement			
unitCost	100	dollars	
weight	30	kg	

The Analysis Viewer shows all components in the first column. The other columns are properties for all stereotypes chosen for this instance. If a property is not part of a stereotype applied to an element, that field is greyed out. You can use the Filter button to hide properties for certain stereotypes. When you select an element, Instance Properties shows its stereotypes and property values. You can save an instance in a MAT-file, and open it again in the Analysis Viewer.



Update Element
Update the values of the selected element

Update Model
Update all the values in the source model

Instances	Cost	Weight	volume	devCost
systemWithProps				
Computer				
PowerSource				
Robot				

INSTANCE PROPERTIES

ComponentInstance: PowerSource

Property	Value	Units	Edit
SystemProfile.PhysicalComponent			
volume	0	m3	
SystemProfile.PhysicalElement			
unitCost	100	dollars	
weight	30	kg	

If you make changes in the model while an instance is open, you can synchronize the instance with the model. **Update** pushes the changes from the instance to the model. **Refresh** updates the instance from the model. Unsynchronized changes are shown in a different color. Selecting a single element enables the option to **Update Element**.

Write Analysis Function

Write a function to analyze the architecture model using instance API. Analysis functions are MATLAB functions that compute values necessary to evaluate the architecture using properties of each element in the model instance.

You can add an analysis function as you set up the analysis instance. After you select the stereotypes of interest, create a template function by clicking + next to the **Analysis function** field. The generated M-file includes the code to obtain all property values from all stereotypes that are subject to analysis. The analysis function operates on a single element — aggregate values are generated by iterating this function over all elements in the model when you run the analysis from Analysis Viewer.

```
function systemWithProps_1(instance,varargin)

if instance.isComponent()
    if instance.hasValue('SystemProfile.PhysicalElement.unitCost')
        sysComponent_unitPrice = instance.getValue('SystemProfile.PhysicalElement.unitCost');
    end
    for child = instance.Components
        comp_price = child.getValue('SystemProfile.PhysicalElement.unitCost');
        sysComponent_unitPrice = sysComponent_unitPrice + comp_price;
    end
```

```
   instance.setValue('SystemProfile.PhysicalElement.unitCost',sysComponent_unitPrice);
end
```

In the generated file, `instance` is the instance of the element on which the analysis function runs currently. You can perform these operations for analysis:

- Access a property of the instance: `instance.getValue('<profile>.<stereotype>.<property>')`
- Set a property of an instance: `instance.setValue('<profile>.<stereotype>.<property>',value)`
- Access the subcomponents of a component: `instance.Components`
- Access the connectors in component: `instance.Connectors`

The `getValue` function generates an error if the property does not exist. You can use `hasValue` to query whether elements in the model have the properties before getting the value.

As an example, this code computes the weight of a component as a sum of the weights of its subcomponents.

```
if instance.isComponent()
    if instance.hasValue('SystemProfile.PhysicalElement.weight')
        weight = instance.getValue('SystemProfile.PhysicalElement.weight');
    end
  for child = instance.Components
    subcomp_weight = child.getValue('SystemProfile.PhysicalElement.weight');
    weight = weight + subcomp_weight;
  end
  instance.setValue('SystemProfile.PhysicalElement.weight',weight)
end
```

Once the analysis function is complete, add it to the analysis under the **Analysis function** box. An analysis function can take additional input arguments, for example, a conversion constant if the weights are in different units in different stereotypes. When this code runs for all components recursively, starting from the deepest components in the hierarchy to the top level, the overall weight of the system is assigned to the `weight` property of the top-level component.

Run Analysis Function

Run an analysis function using the Analysis Viewer.

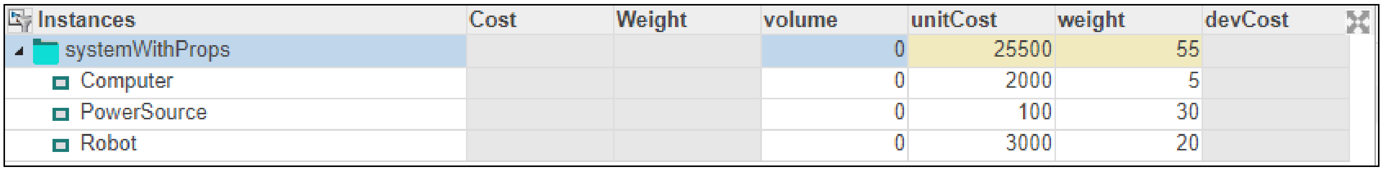
1. Select or change the analysis function using the **Analyze** menu.
2. Select the iteration method.

    - `Pre-order` — Start from the top level, move to a child component, process the subcomponents of that component recursively before moving to a sibling component.
    - `Top-Down` — Like pre-order, but process all sibling components before moving to their subcomponents.
    - `Post-order` — Start from components with no subcomponents, process each sibling and then move to parent.
    - `Bottom-up` — Like post-order, but process all subcomponents at the same depth before moving to their parents.

    The iteration method depends on what kind of analysis is to be run. For example, for an analysis where the component weight is the sum of the weights of its components, you must make sure the subcomponent weights are computed first, so the iteration method must be bottom-up.

3. Click the **Analyze** button.

System Composer runs the analysis function over each model element and computes results. The computed properties are shown in a different color in the Analysis Viewer.



Instances	Cost	Weight	volume	unitCost	weight	devCost
systemWithProps			0	25500	55	
Computer			0	2000	5	
PowerSource			0	100	30	
Robot			0	3000	20	

See Also

`deleteInstance` | `getValue` | `hasValue` | `instantiate` | `iterate` | `loadInstance` | `lookup` | `refresh` | `save` | `setValue` | `systemcomposer.analysis.Instance` | `update`

More About

- “Define Profiles and Stereotypes” on page 4-2
- “Use Stereotypes and Profiles” on page 4-10
- “Simulating Mobile Robot with System Composer Workflow” on page 6-46

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

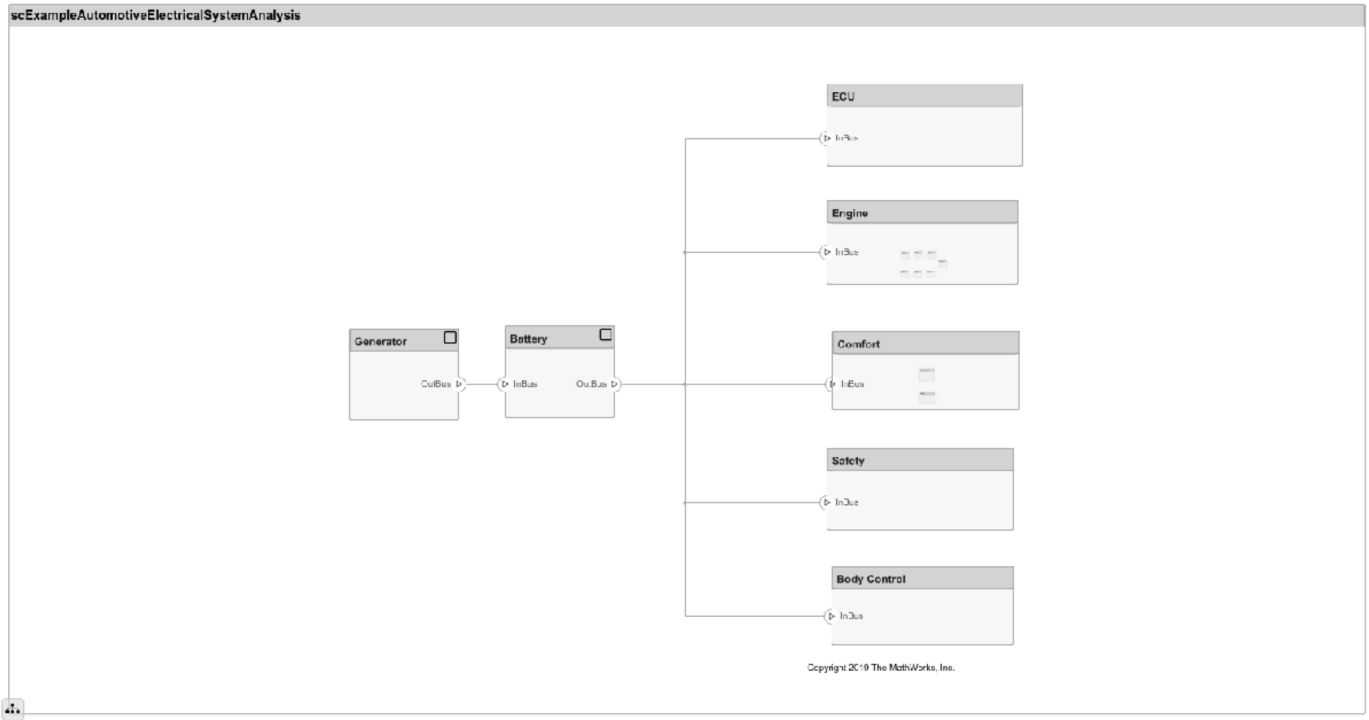
The generator charges the battery while the engine is running. The battery, along with the generator supports the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the `InRushCurrent` stereotype property defined. Based on the properties 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');
```

See Also

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

More About

- “Analyze Architecture” on page 6-10

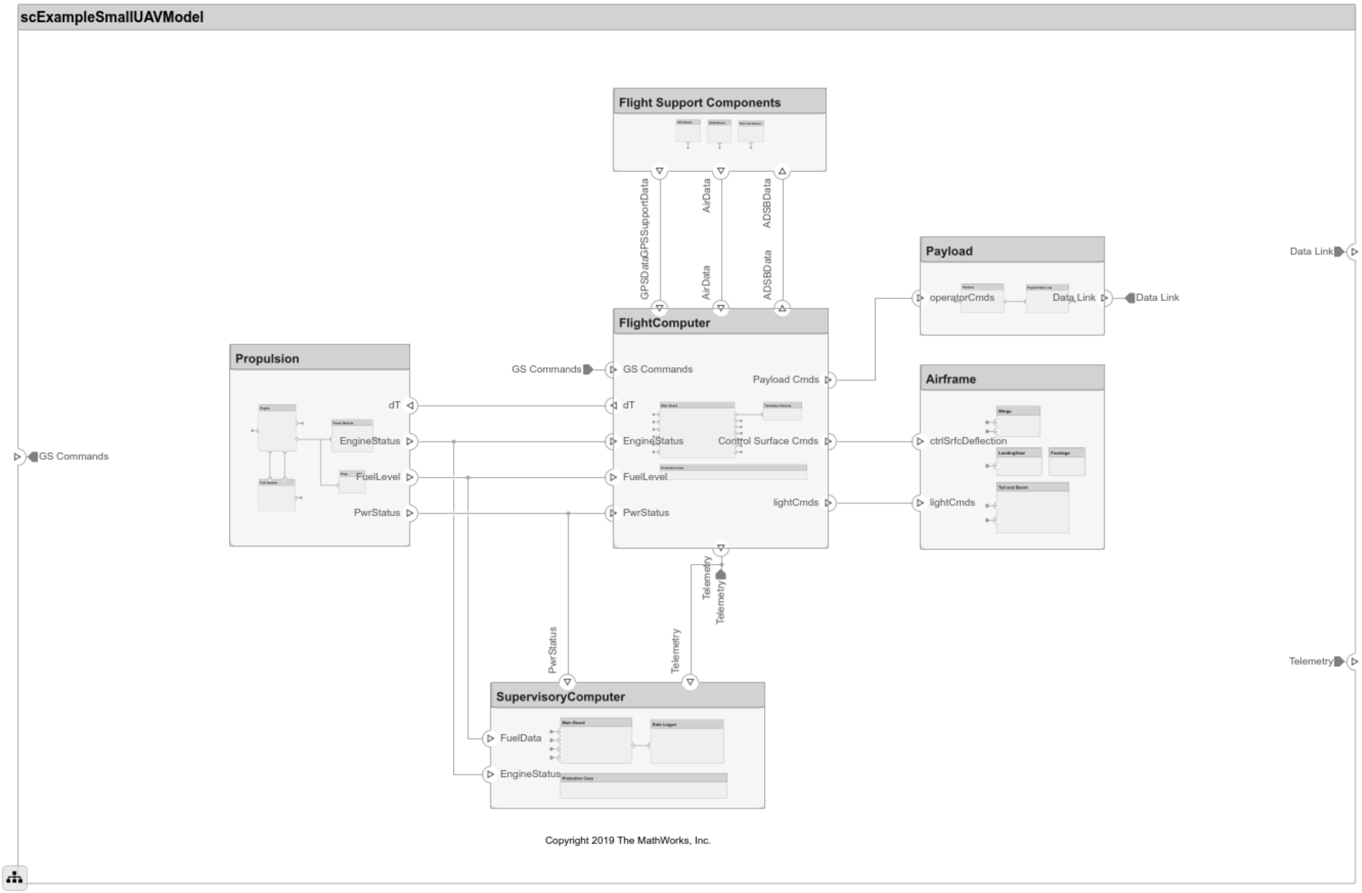
Modeling System Architecture of Small UAV

Overview

This example shows how to use System Composer to set up the architecture for a small unmanned aerial vehicle, composed of six top-level components. Learn how to refine your architecture design by authoring interfaces, inspect linked textual requirements, define profiles and stereotypes, and run a static analysis on such an architecture model.

Open the project.

```
>> scExampleSmallUAV
```

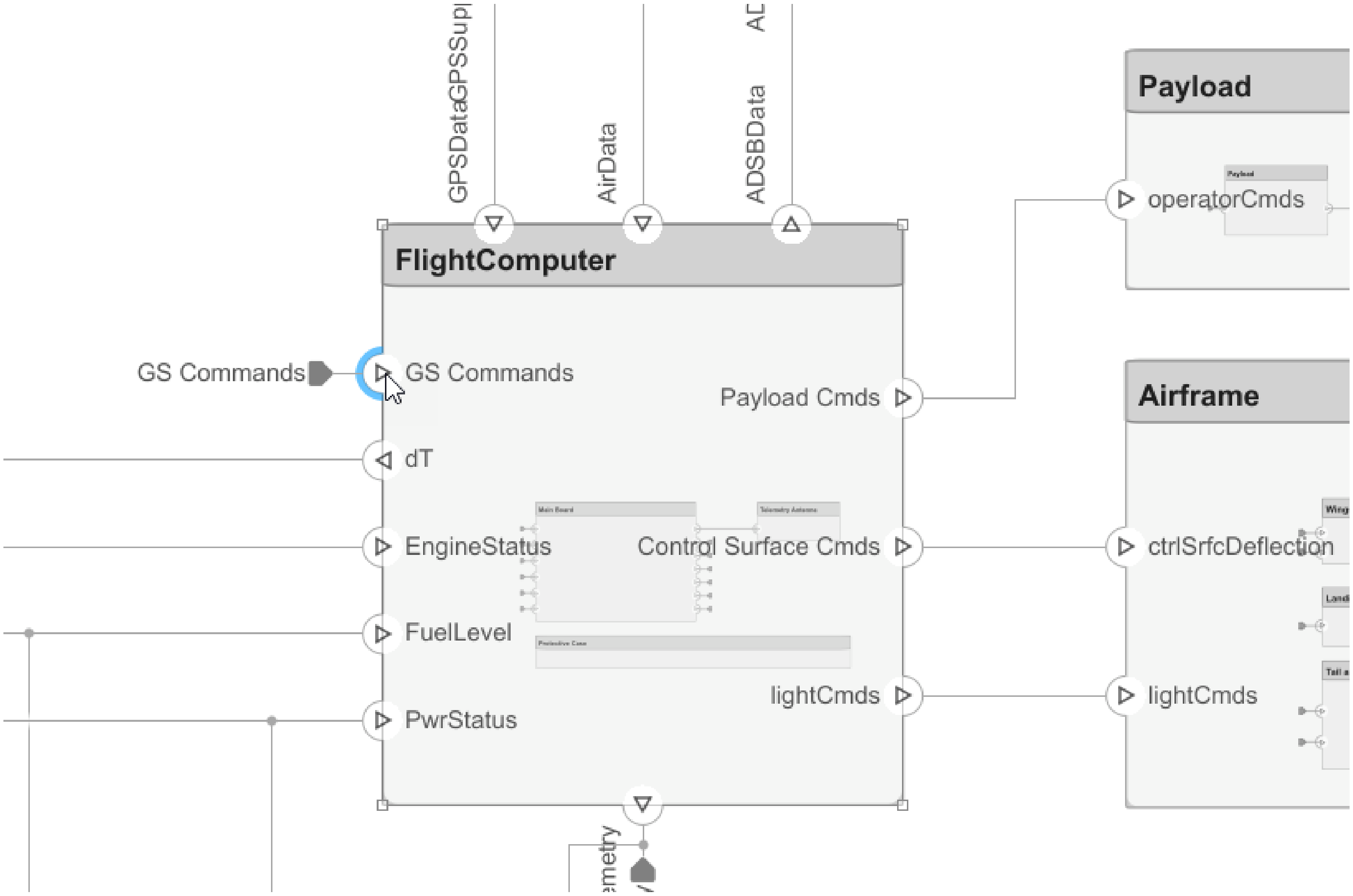


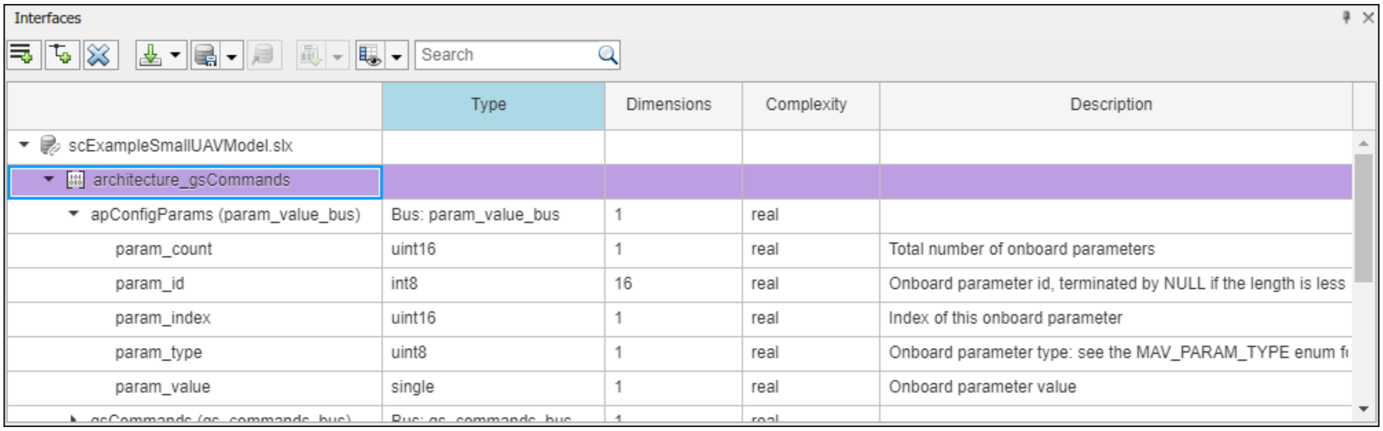
Each top-level component is decomposed into its subcomponents. Navigate through the hierarchy to view the composition for each component. The root component, `scExampleSmallUAVModel`, has input and output ports that represent data exchange between the system and its environment.

Author Interfaces

Define interfaces for domain-specific data between connections. The information shared between two ports defined by interface element property values further enhances the specification. In the **Modeling** tab in the toolstrip, select **Design**, then click **Interface Editor**.

Click the **GS Commands** port on the architecture model to highlight the **architecture_gsCommands** interface and indicate the assignment of the interface.



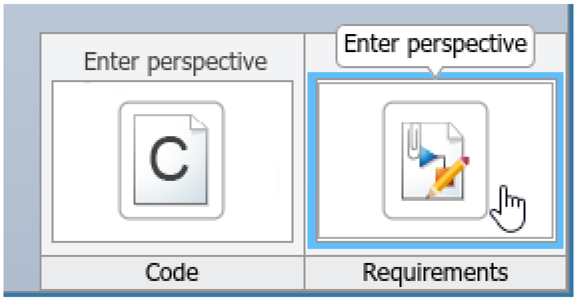


	Type	Dimensions	Complexity	Description
scExampleSmallUAVModel.slx				
architecture_gsCommands				
apConfigParams (param_value_bus)	Bus: param_value_bus	1	real	
param_count	uint16	1	real	Total number of onboard parameters
param_id	int8	16	real	Onboard parameter id, terminated by NULL if the length is less
param_index	uint16	1	real	Index of this onboard parameter
param_type	uint8	1	real	Onboard parameter type: see the MAV_PARAM_TYPE enum f
param_value	single	1	real	Onboard parameter value
gsCommands (gs_commands_bus)	Bus: gs_commands_bus	1	real	

Inspect Requirements

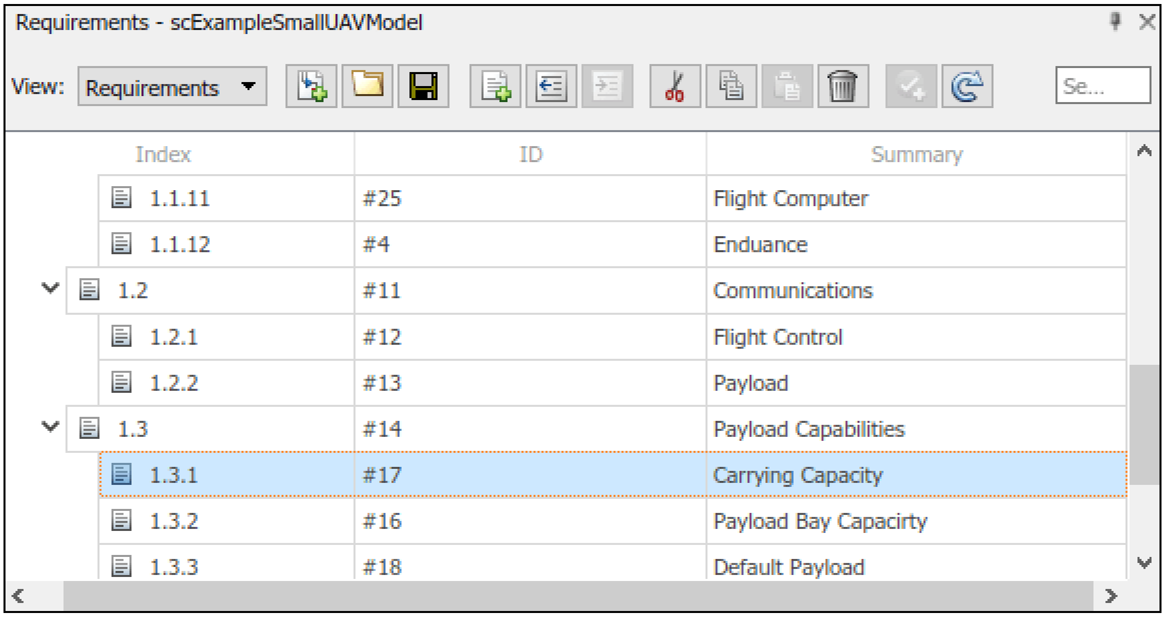
A Simulink Requirements license is required to inspect requirements in a System Composer architecture model.

Components in the architecture model link to system requirements defined in `smallUAVReqs.slreqx`. Open the **Requirements Perspective**. In the bottom right corner of the model pane, click **Show Perspectives**. Then, click **Requirements**.



Select components on the model to see the requirement they link to, or, conversely, select items in the **Requirements** view to see which components implement them. Requirements can also be linked to connectors or ports to allow traceability throughout your design artifacts. To edit the requirements in `smallUAVReqs.slreqx`, select the **Requirements Editor** from the menu.

The `Carrying Capacity` requirement highlights the total mass able to be carried by the aircraft. This requirement, along with the weight of the aircraft, is part of the mass rollup analysis performed for early verification and validation.



Index	ID	Summary
1.1.11	#25	Flight Computer
1.1.12	#4	Enduance
1.2	#11	Communications
1.2.1	#12	Flight Control
1.2.2	#13	Payload
1.3	#14	Payload Capabilities
1.3.1	#17	Carrying Capacity
1.3.2	#16	Payload Bay Capacirty
1.3.3	#18	Default Payload

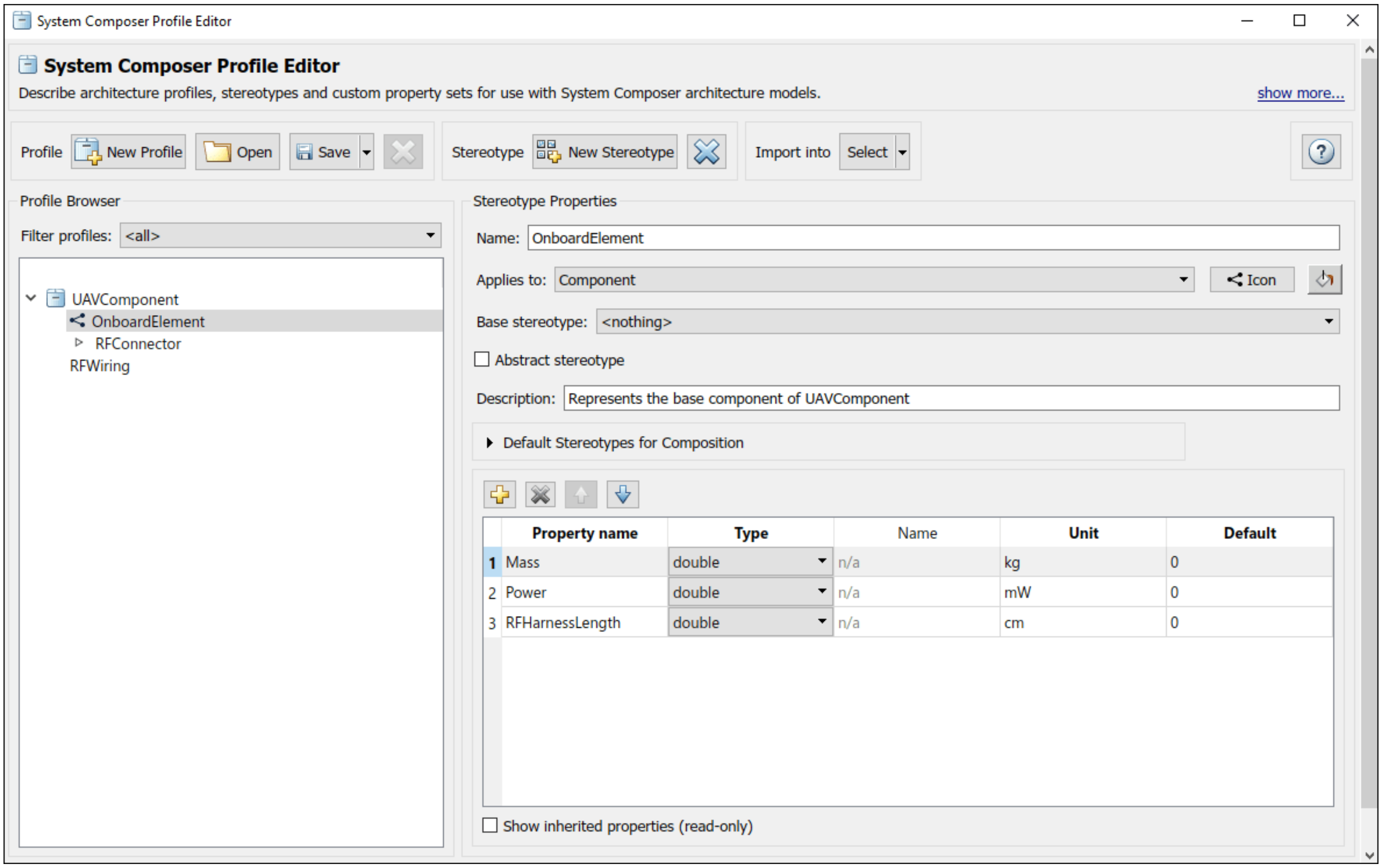
Define Profiles and Stereotypes

To complete specifications and enable analysis later in the design process, stereotypes add custom metadata to architecture model elements. This model has stereotypes for these elements:

- On-board element, applicable to components
- RF connector, applicable to ports
- RF wiring, applicable to connectors

Stereotypes are defined in `.xml` files by using Profiles. The profile `UAVComponent.xml` is attached to this model. Edit a profile by using the **Profile Editor**. On the **Modeling** tab, click **Import > Edit**.

The display appears below.



System Composer Profile Editor

Describe architecture profiles, stereotypes and custom property sets for use with System Composer architecture models.

Profile Browser

Filter profiles: <all>

- UAVComponent
  - OnboardElement
  - RFConnector
- RFWiring

Stereotype Properties

Name: OnboardElement

Applies to: Component

Base stereotype: <nothing>

□ Abstract stereotype

Description: Represents the base component of UAVComponent

▸ Default Stereotypes for Composition

	Property name	Type	Name	Unit	Default
1	Mass	double	n/a	kg	0
2	Power	double	n/a	mW	0
3	RFHarnessLength	double	n/a	cm	0

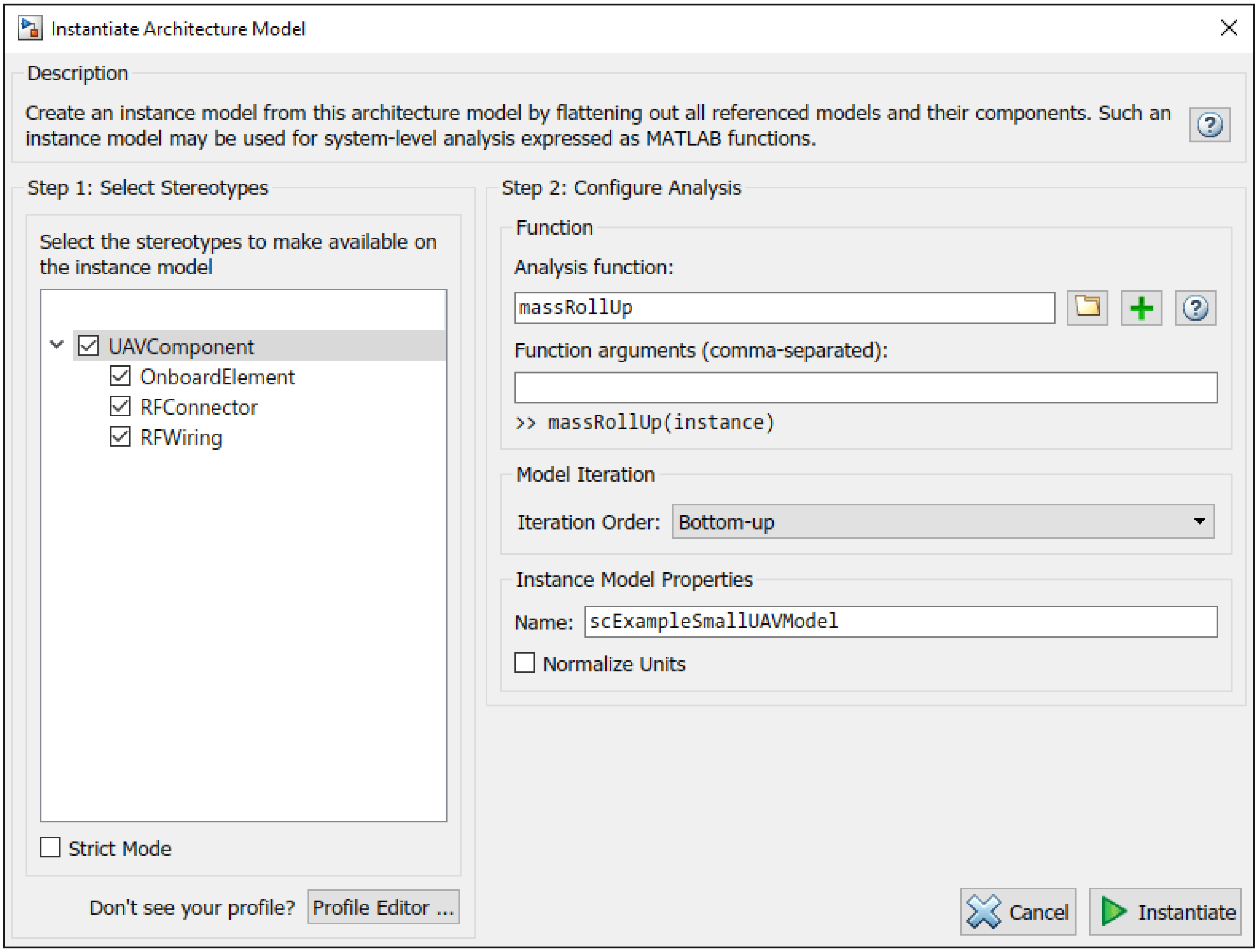
□ Show inherited properties (read-only)

Analyze the Model

To run static analyses on your system, create an Analysis Model from your architecture model. An Analysis Model is a tree of instances generated from the elements of the architecture model in which all referenced models are flattened out, and all variants are resolved.

Click **Analysis Model** on the **Views** menu.

Run a mass rollup on this model. In the dialog, select the stereotypes that you want to include in your analysis. Select the analysis function by browsing to `utilities/massRollUp.m`. Set the model iteration mode to **Bottom-up**.



Instantiate Architecture Model

Description

Create an instance model from this architecture model by flattening out all referenced models and their components. Such an instance model may be used for system-level analysis expressed as MATLAB functions.

Step 1: Select Stereotypes

Select the stereotypes to make available on the instance model

- ☑ UAVComponent
  - ☑ OnboardElement
  - ☑ RFConnector
  - ☑ RFWiring

□ Strict Mode

Don't see your profile? Profile Editor ...

Step 2: Configure Analysis

Function

Analysis function: massRollUp

Function arguments (comma-separated):

```
>> massRollUp(instance)
```

Model Iteration

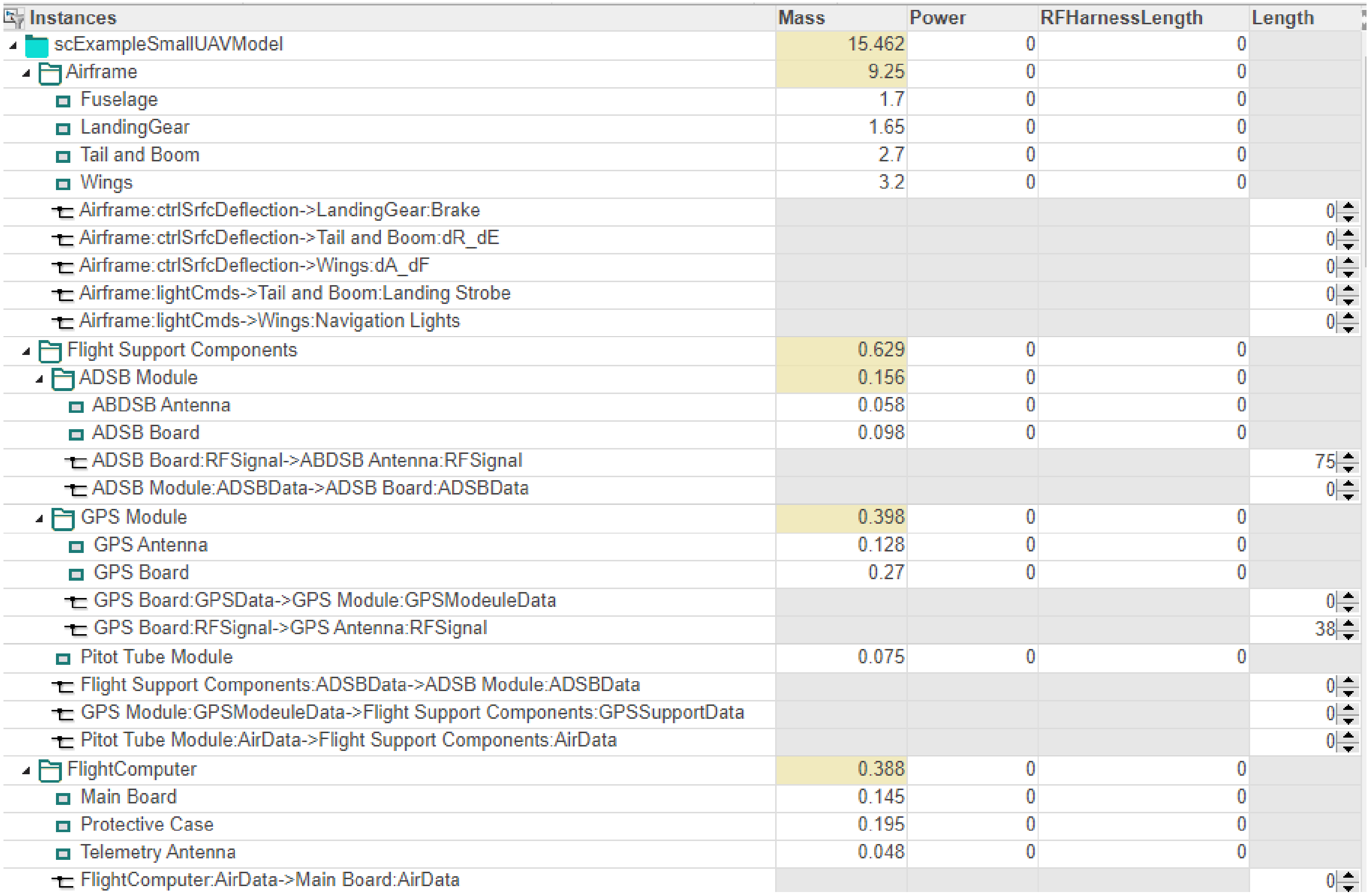
Iteration Order: Bottom-up

Instance Model Properties

Name: scExampleSmallUAVModel

□ Normalize Units

Uncheck `Strict Mode` so that all components can have a `Mass` property instantiated to facilitate calculation of total mass. Click **Instantiate** to generate an analysis.



Instances	Mass	Power	RFHarnessLength	Length
scExampleSmallUAVModel	15.462	0	0	
Airframe	9.25	0	0	
Fuselage	1.7	0	0	
LandingGear	1.65	0	0	
Tail and Boom	2.7	0	0	
Wings	3.2	0	0	
Airframe:ctrlSrfcDeflection->LandingGear:Brake				0
Airframe:ctrlSrfcDeflection->Tail and Boom:dR_dE				0
Airframe:ctrlSrfcDeflection->Wings:dA_dF				0
Airframe:lightCmds->Tail and Boom:Landing Strobe				0
Airframe:lightCmds->Wings:Navigation Lights				0
Flight Support Components	0.629	0	0	
ADSB Module	0.156	0	0	
ABDSB Antenna	0.058	0	0	
ADSB Board	0.098	0	0	
ADSB Board:RFSignal->ABDSB Antenna:RFSignal				75
ADSB Module:ADSBData->ADSB Board:ADSBData				0
GPS Module	0.398	0	0	
GPS Antenna	0.128	0	0	
GPS Board	0.27	0	0	
GPS Board:GPSData->GPS Module:GPSModeuleData				0
GPS Board:RFSignal->GPS Antenna:RFSignal				38
Pitot Tube Module	0.075	0	0	
Flight Support Components:ADSBData->ADSB Module:ADSBData				0
GPS Module:GPSModeuleData->Flight Support Components:GPSSupportData				0
Pitot Tube Module:AirData->Flight Support Components:AirData				0
FlightComputer	0.388	0	0	
Main Board	0.145	0	0	
Protective Case	0.195	0	0	
Telemetry Antenna	0.048	0	0	
FlightComputer:AirData->Main Board:AirData				0

Once on the **Analysis Viewer** screen, click **Analyze**. The analysis function iterates through model elements bottom up, assigning the `Mass` property of each component as a sum of the `Mass` properties of its subcomponents. The overall weight of the system is assigned to the `Mass` property of the top level component, `scExampleSmallUAVModel`.

See Also

`addProperty` | `addStereotype` | `applyStereotype` | `createProfile` | `instantiate` | `setInterface`

More About

- “Define Interfaces” on page 3-2
- “Assign Interfaces to Ports” on page 3-7
- “Save, Link, and Delete Interfaces” on page 3-12
- “Link and Trace Requirements” on page 6-25
- “Manage Requirements” on page 2-2
- “Define Profiles and Stereotypes” on page 4-2
- “Use Stereotypes and Profiles” on page 4-10
- “Analyze Architecture” on page 6-10

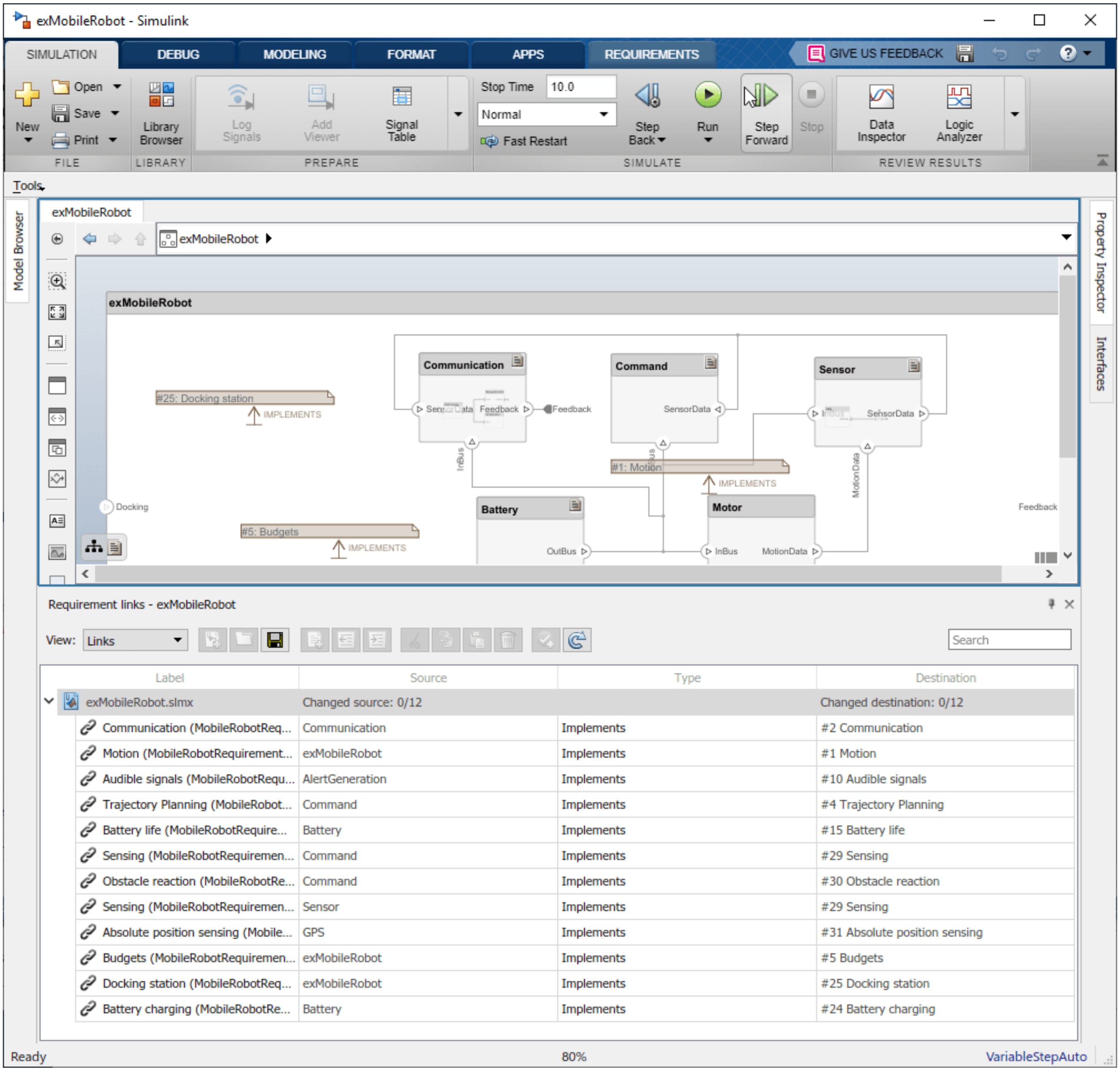
Link and Trace Requirements

This example shows how to work with requirements in an architecture model.

Allocate functional requirements to components to establish traceability. By creating a link between a component and the related requirement, you can track whether all requirements are represented in the architecture. You can also keep requirements and design in sync, for example, if a requirement changes or if the design warrants a revision of the requirements. You can link components to requirements in Simulink® Requirements™, test cases in Simulink Test™, or selections in MATLAB®, Microsoft® Excel®, or Microsoft Word.

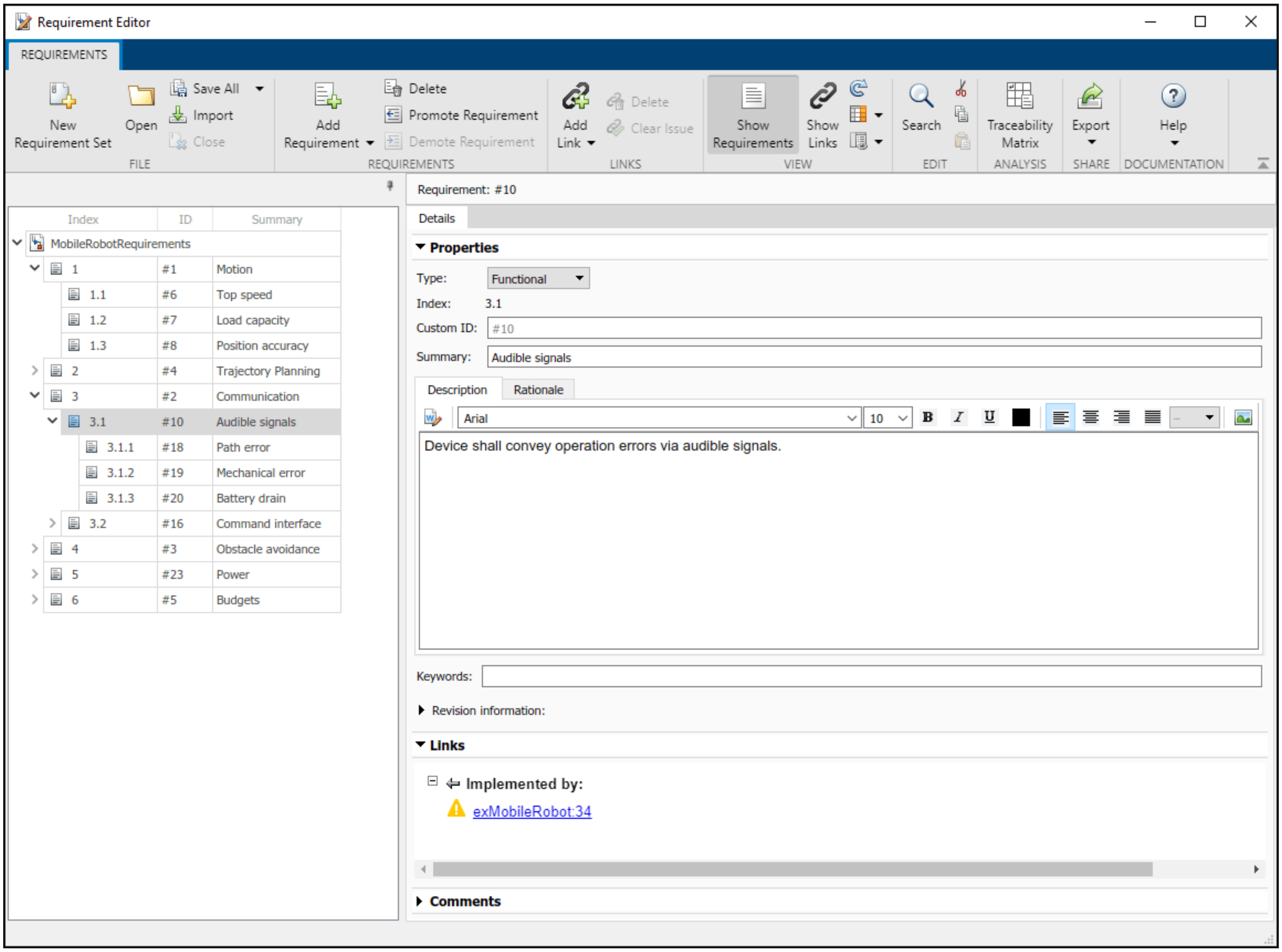
Open the model `exMobileRobot`.

```
open_system('exMobileRobot')
```



Label	Source	Type	Destination
exMobileRobot.slmx	Changed source: 0/12		Changed destination: 0/12
Communication (MobileRobotReq...	Communication	Implements	#2 Communication
Motion (MobileRobotRequirement...	exMobileRobot	Implements	#1 Motion
Audible signals (MobileRobotRequ...	AlertGeneration	Implements	#10 Audible signals
Trajectory Planning (MobileRobot...	Command	Implements	#4 Trajectory Planning
Battery life (MobileRobotRequire...	Battery	Implements	#15 Battery life
Sensing (MobileRobotRequiremen...	Command	Implements	#29 Sensing
Obstacle reaction (MobileRobotRe...	Command	Implements	#30 Obstacle reaction
Sensing (MobileRobotRequiremen...	Sensor	Implements	#29 Sensing
Absolute position sensing (Mobile...	GPS	Implements	#31 Absolute position sensing
Budgets (MobileRobotRequiremen...	exMobileRobot	Implements	#5 Budgets
Docking station (MobileRobotReq...	exMobileRobot	Implements	#25 Docking station
Battery charging (MobileRobotRe...	Battery	Implements	#24 Battery charging

Open the requirements `MobileRobotRequirements.slreqx` in the **Requirements Editor**. The requirements file must be on the MATLAB path. Select the requirement to be linked.



Index	ID	Summary
MobileRobotRequirements		
1	#1	Motion
1.1	#6	Top speed
1.2	#7	Load capacity
1.3	#8	Position accuracy
2	#4	Trajectory Planning
3	#2	Communication
3.1	#10	Audible signals
3.1.1	#18	Path error
3.1.2	#19	Mechanical error
3.1.3	#20	Battery drain
3.2	#16	Command interface
4	#3	Obstacle avoidance
5	#23	Power
6	#5	Budgets

Requirement: #10

Type: Functional

Index: 3.1

Custom ID: #10

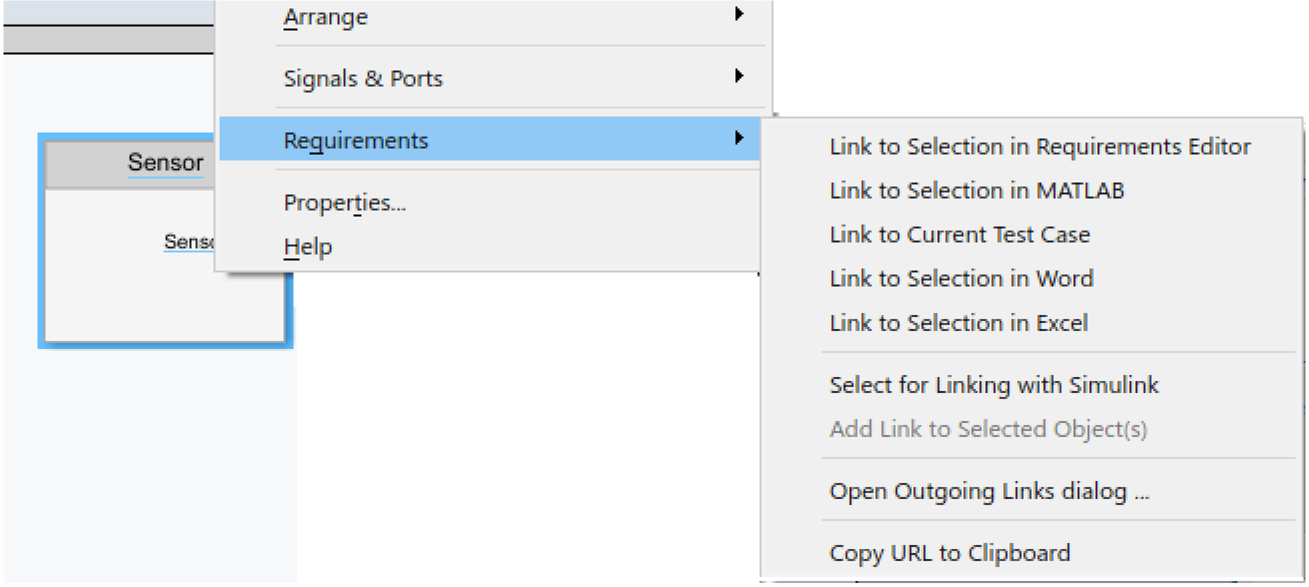
Summary: Audible signals

Device shall convey operation errors via audible signals.

Implemented by:

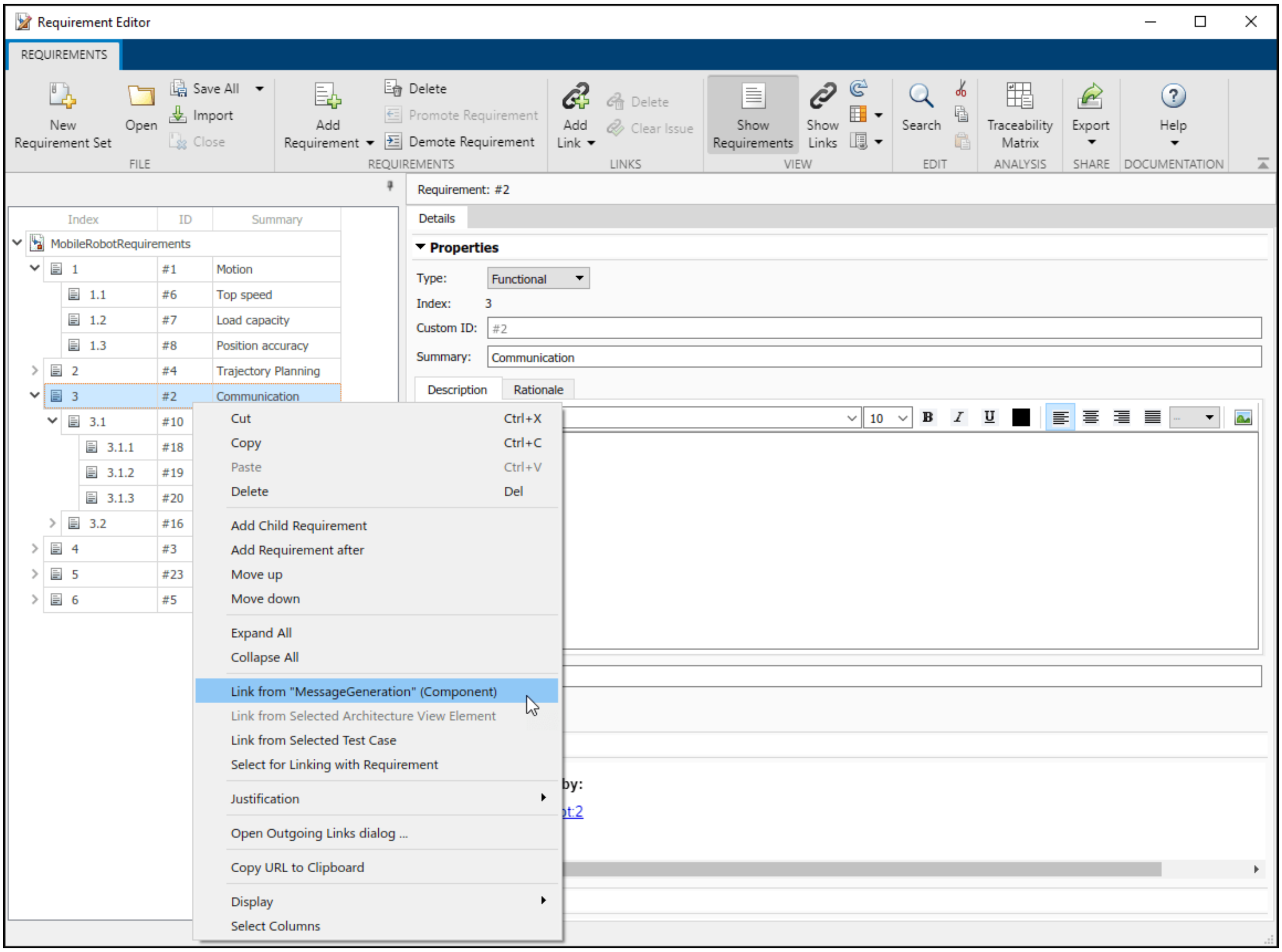
exMobileRobot:34

Select the component to be linked in the architecture model. Right-click and select **Requirements > Link to Selection in Requirements Editor**.


Arrange
Signals & Ports
Requirements
Properties...
Help
Sensor
Sens
Link to Selection in Requirements Editor
Link to Selection in MATLAB
Link to Current Test Case
Link to Selection in Word
Link to Selection in Excel
Select for Linking with Simulink
Add Link to Selected Object(s)
Open Outgoing Links dialog ...
Copy URL to Clipboard

When you first link a requirement in an architecture model, a link set file with extension `.slmx` is created to store requirement links. The **Requirements** context menu displays the linked requirements.

You can also create a link using the Requirements Editor. First, select the component in the architecture model. Then, in the Requirements Editor, right-click the requirement and select **Link from <component_name> (Component)**.



Index	ID	Summary
MobileRobotRequirements		
1	#1	Motion
1.1	#6	Top speed
1.2	#7	Load capacity
1.3	#8	Position accuracy
2	#4	Trajectory Planning
3	#2	Communication
3.1	#10	
3.1.1	#18	
3.1.2	#19	
3.1.3	#20	
3.2	#16	
4	#3	
5	#23	
6	#5	

Requirement: #2

Type: Functional

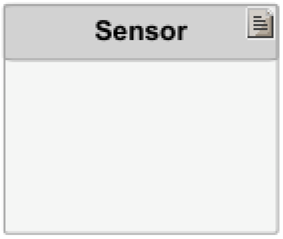
Index: 3

Custom ID: #2

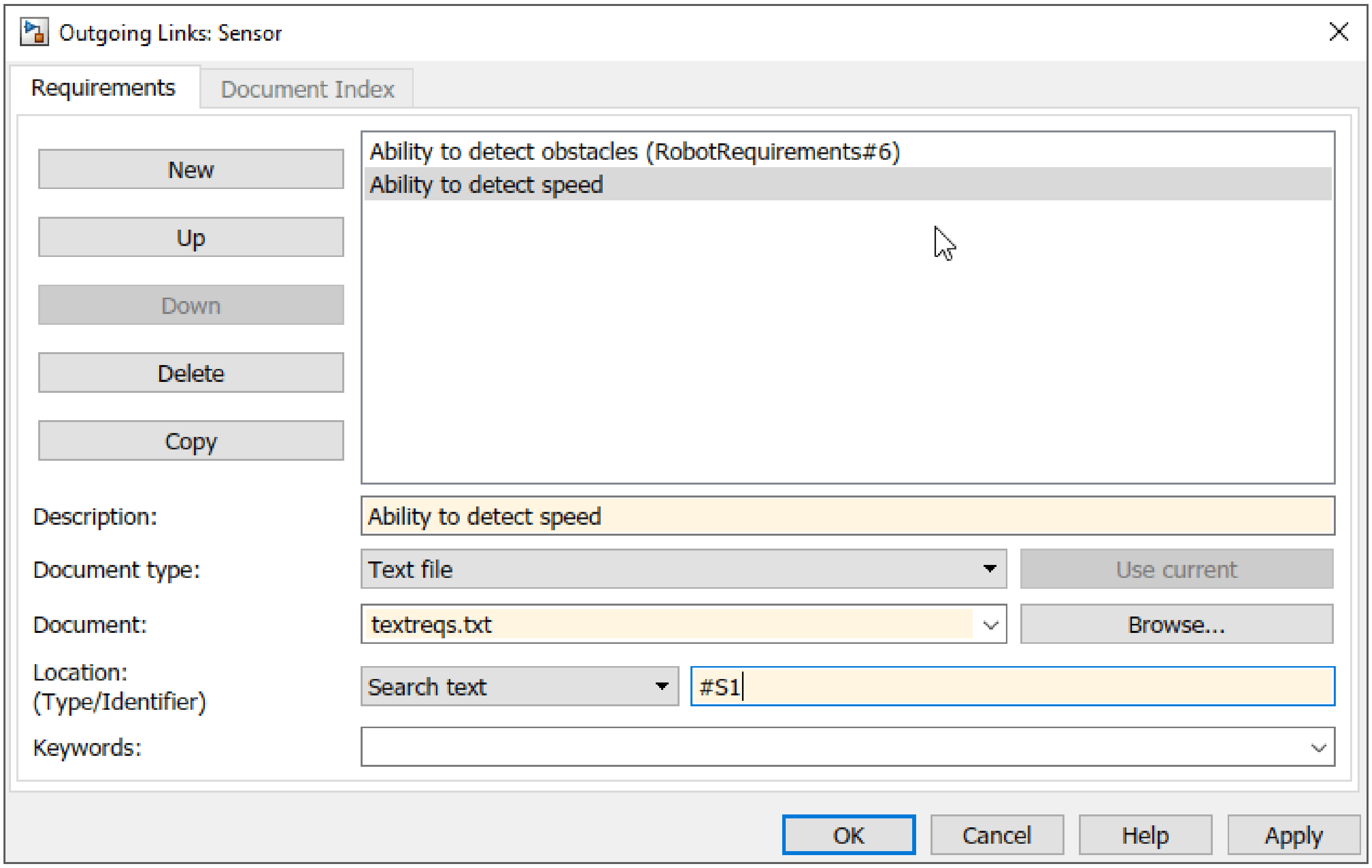
Summary: Communication

You can also create requirement links with blocks and subsystems in Simulink models. for more information, see “Link Blocks and Requirements” (Simulink Requirements).

The 📄 badge on a component indicates that it is linked to a requirement. This badge also shows at the lower-left corner of the architecture model.



To trace requirement links to a component, right-click and select **Requirements > Open Outgoing Links dialog**. Here, you can create new requirements, delete existing ones, and change their order.



Outgoing Links: Sensor

Ability to detect obstacles (RobotRequirements#6)
Ability to detect speed

Description: Ability to detect speed

Document type: Text file

Document: textreqs.txt

Location: (Type/Identifier) Search text #S1

Keywords:

See Also

`updateLinksToReferenceRequirements`

More About

- “Manage Requirements” on page 2-2
- “Simulating Mobile Robot with System Composer Workflow” on page 6-46
- “View Simulink Requirements Links Associated with Model Elements”

Modeling System Architecture of Keyless Entry System

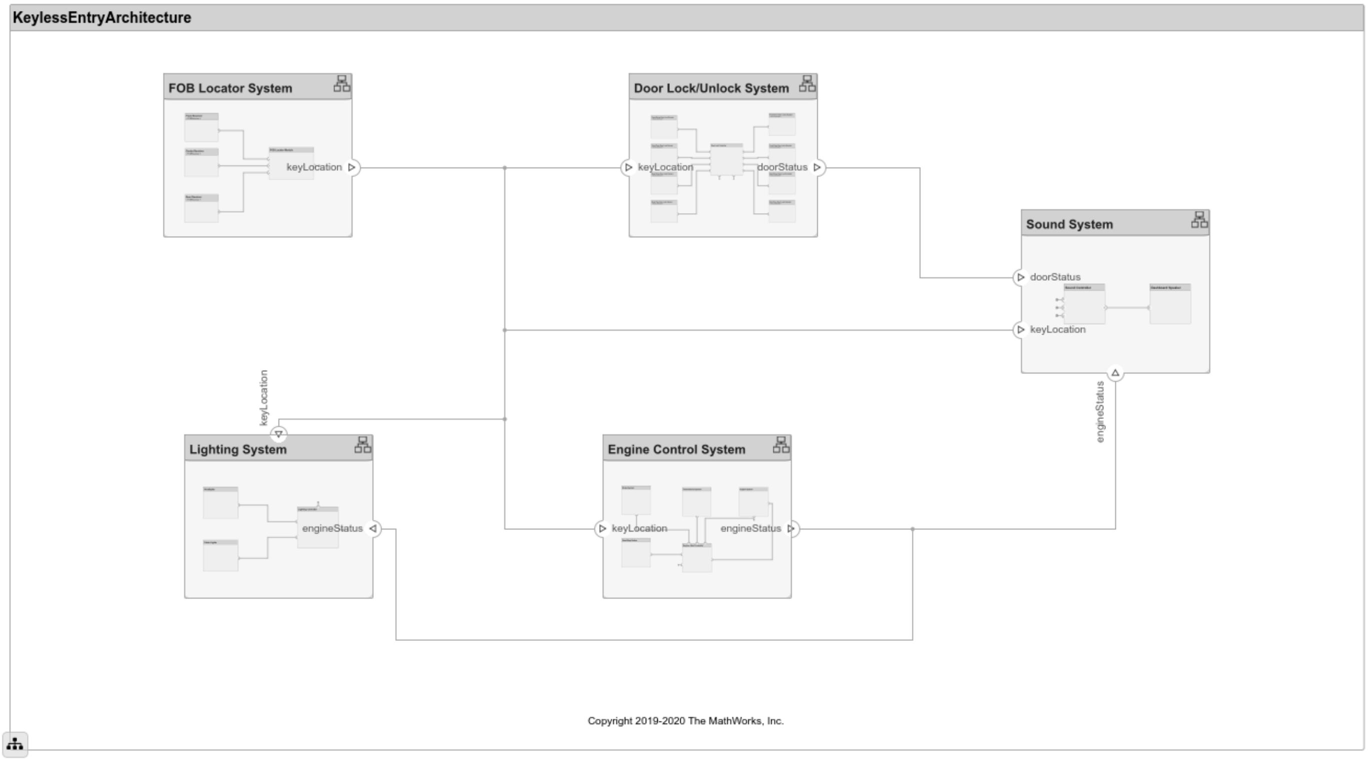
Overview

This example shows how to set up the architecture for a keyless entry system for a vehicle. You also learn how to create different architecture views for different stakeholder concerns.

Open the project.

```
scKeylessEntrySystem
```

```
Starting: Simulink
```



Opening the Architecture Views

You can create, view, and edit architecture views in the Architecture Views editor. To launch the editor, select the **Architecture Views** button from the **Modeling** tab in the toolstrip. Select from one of the existing views for the model. The model has these views:

- Key FOB Position Dataflow — An operational view of the components in the model that are making use of the **KeyFOBPosition** interface.
- Door Lock System Supplier Breakdown — A functional view of the components in the door lock system grouped by which supplier is providing the given components.
- Sound System Supplier Breakdown — A functional view of the components in the sound system grouped by which supplier is providing the given components.

- Software Component Review Status — A physical view of the components in the model with the **SoftwareComponent** stereotype applied grouped by the value of the `ReviewStatus` property.

See Also

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

More About

- “Create Architecture Views Interactively” on page 1-37
- “Create Architectural Views Programmatically” on page 1-45
- “Display Component Hierarchy Using Hierarchy Views” on page 1-58

Extract the Architecture of a Simulink Model Using System Composer

Overview

This example shows how to export an existing Simulink® model to a System Composer™ architecture model. The algorithmic sections of the original model are removed and structural information is preserved during this process. Requirements links, if any, are also preserved.

Converting Simulink Model to System Composer Architecture

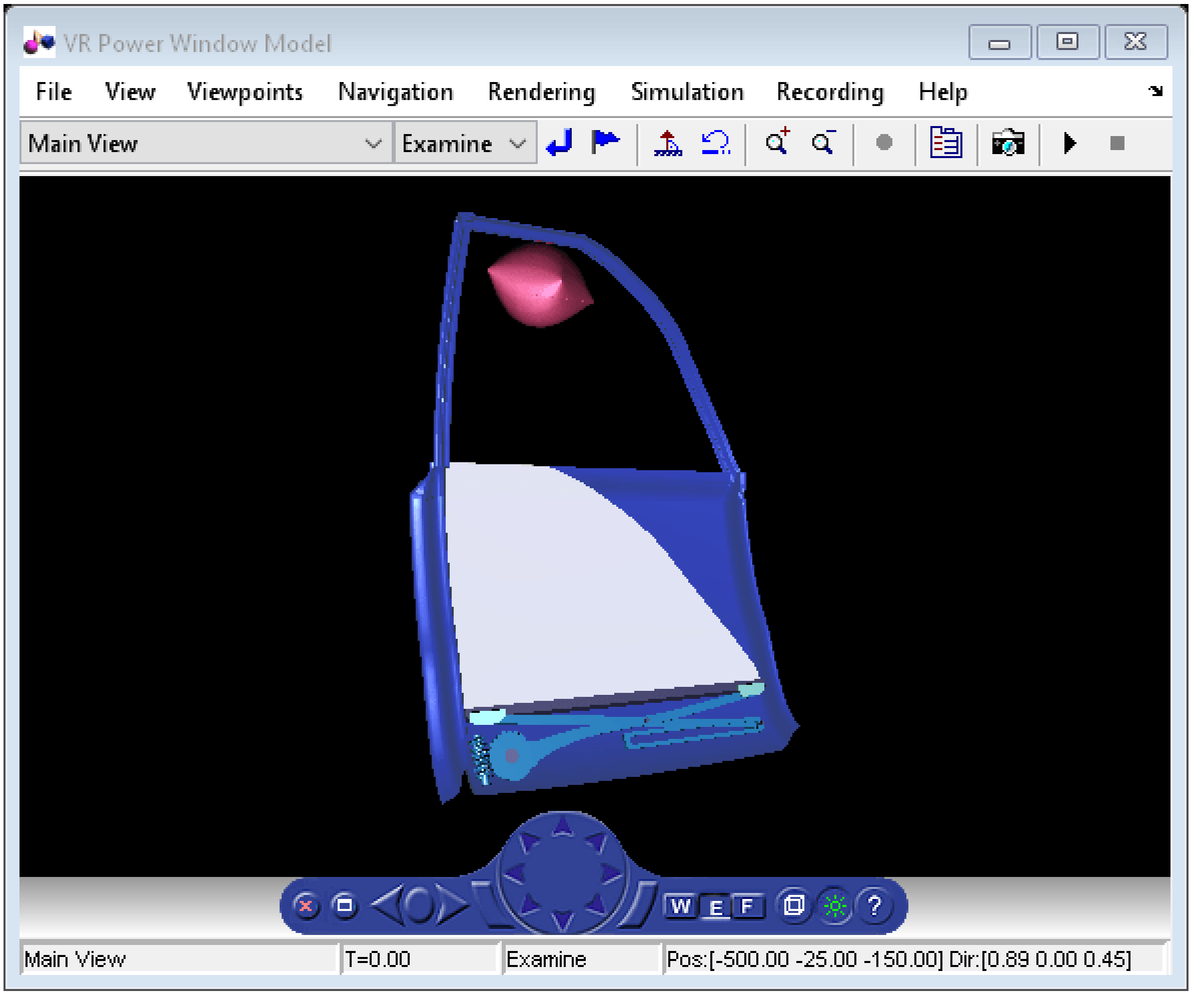
System Composer converts structural constructs in a Simulink model to equivalent architecture model constructs:

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

Open the Model

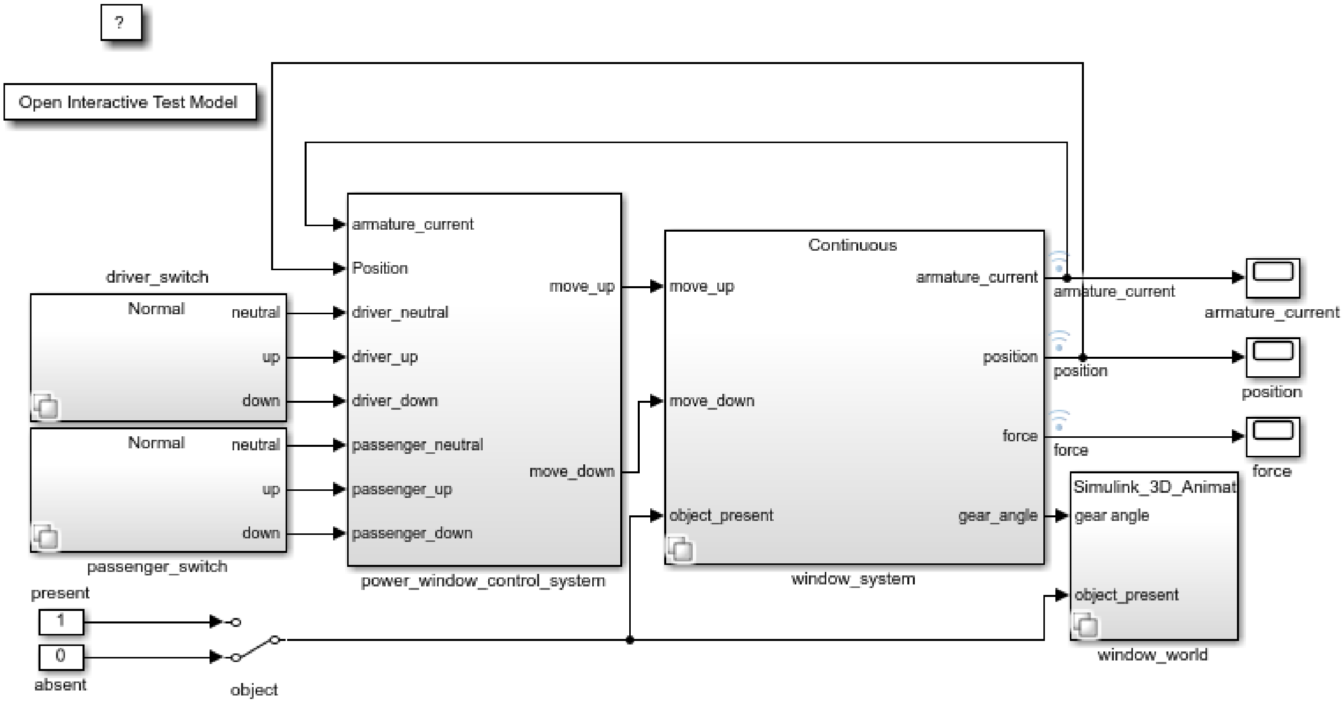
Open the Simulink model of a system.

```
slexPowerWindowStart
```



Main View | T=0.00 | Examine | Pos:[-500.00 -25.00 -150.00] Dir:[0.89 0.00 0.45]

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

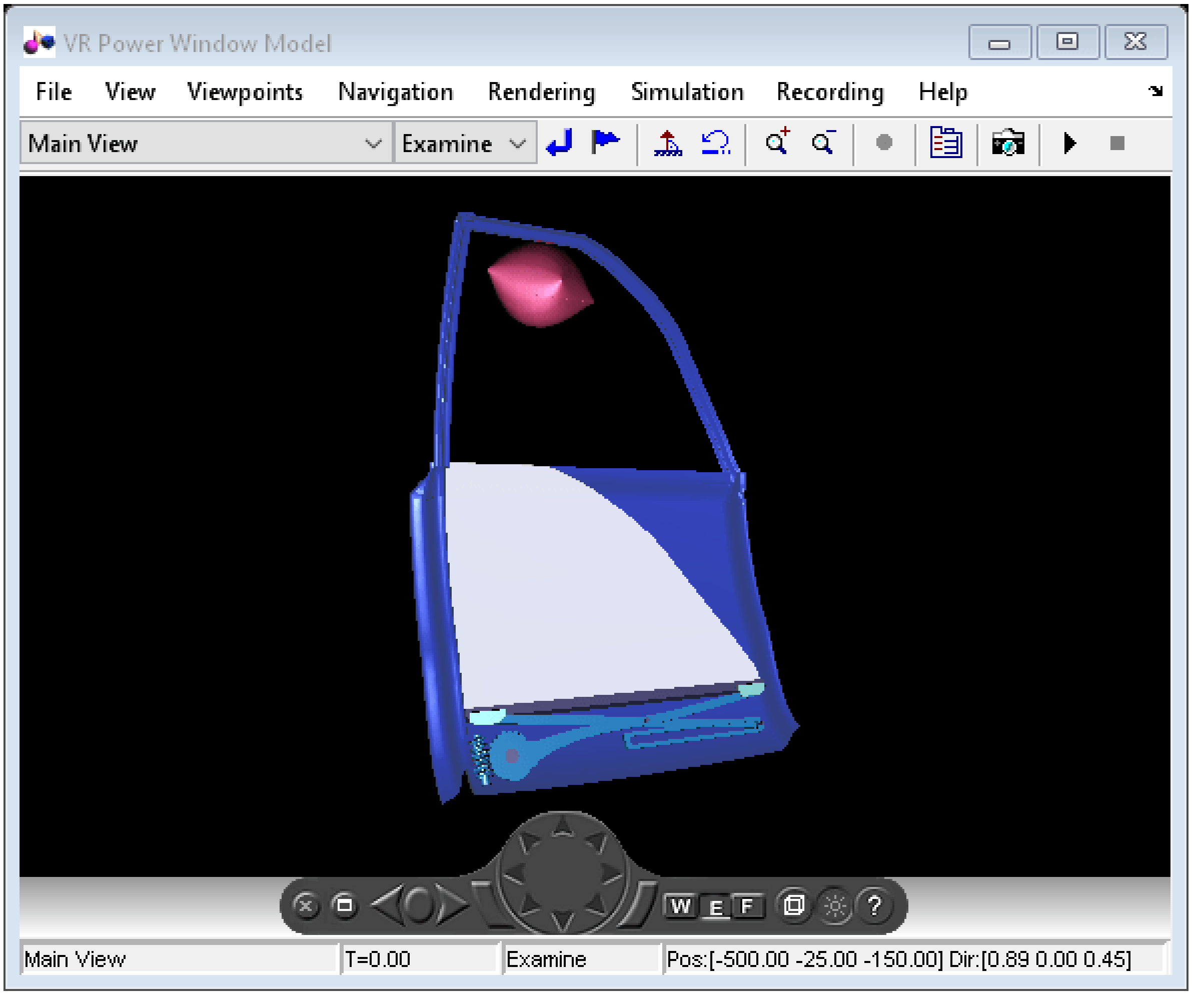


Copyright 2013-2016 The MathWorks, Inc.

Export the Model

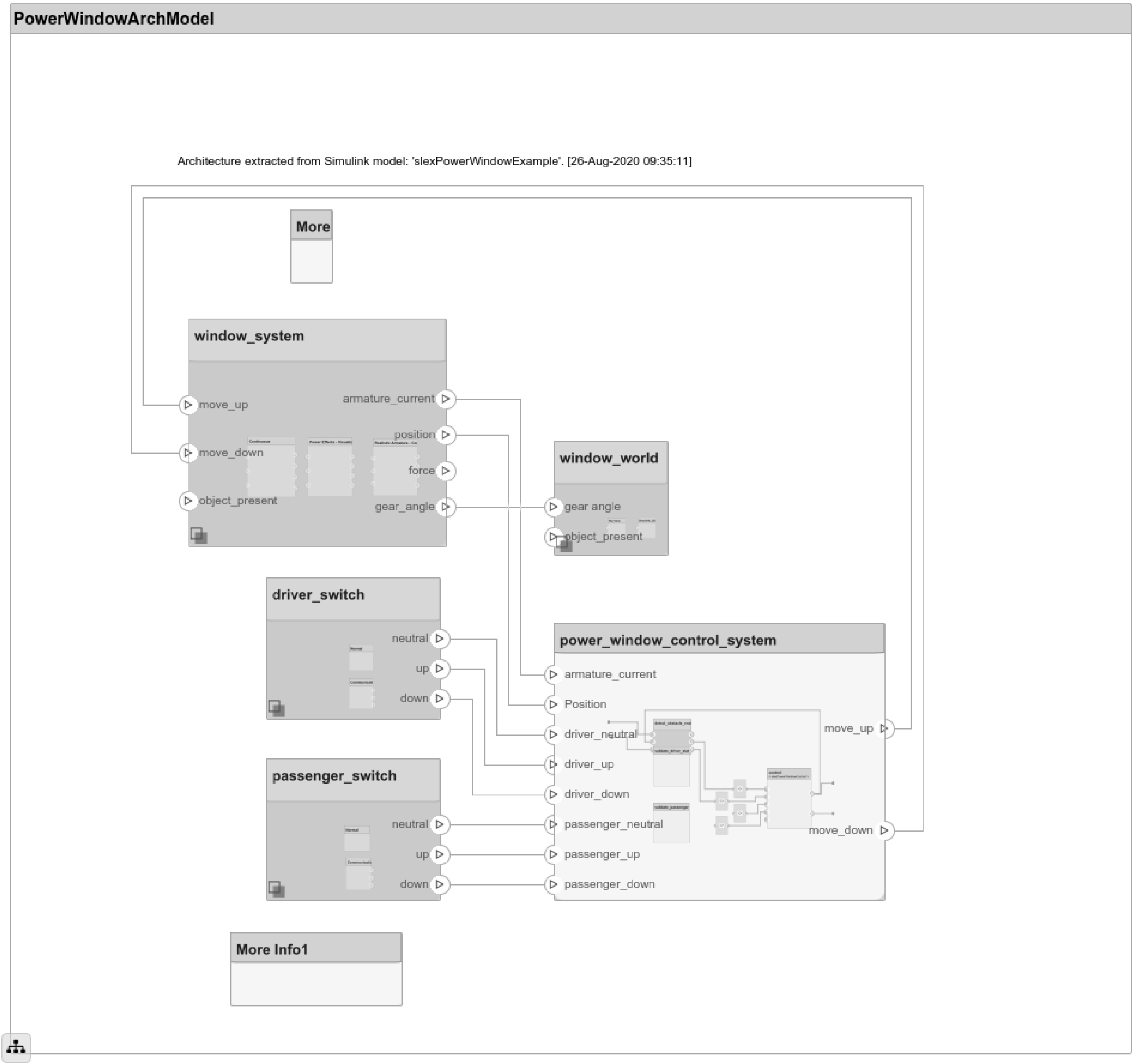
Extract an architecture model from the original model.

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



Main View | T=0.00 | Examine | Pos:[-500.00 -25.00 -150.00] Dir:[0.89 0.00 0.45]

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



Architecture extracted from Simulink model: 'slexPowerWindowExample'. [26-Aug-2020 09:35:11]

Simulate this Example

To simulate this example from start to finish, run in the Command Window:

```
scExamplePowerWindowBottomUp
```

See Also

`extractArchitectureFromSimulink`

More About

- “Extract Architecture from Simulink Model” on page 5-12

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.

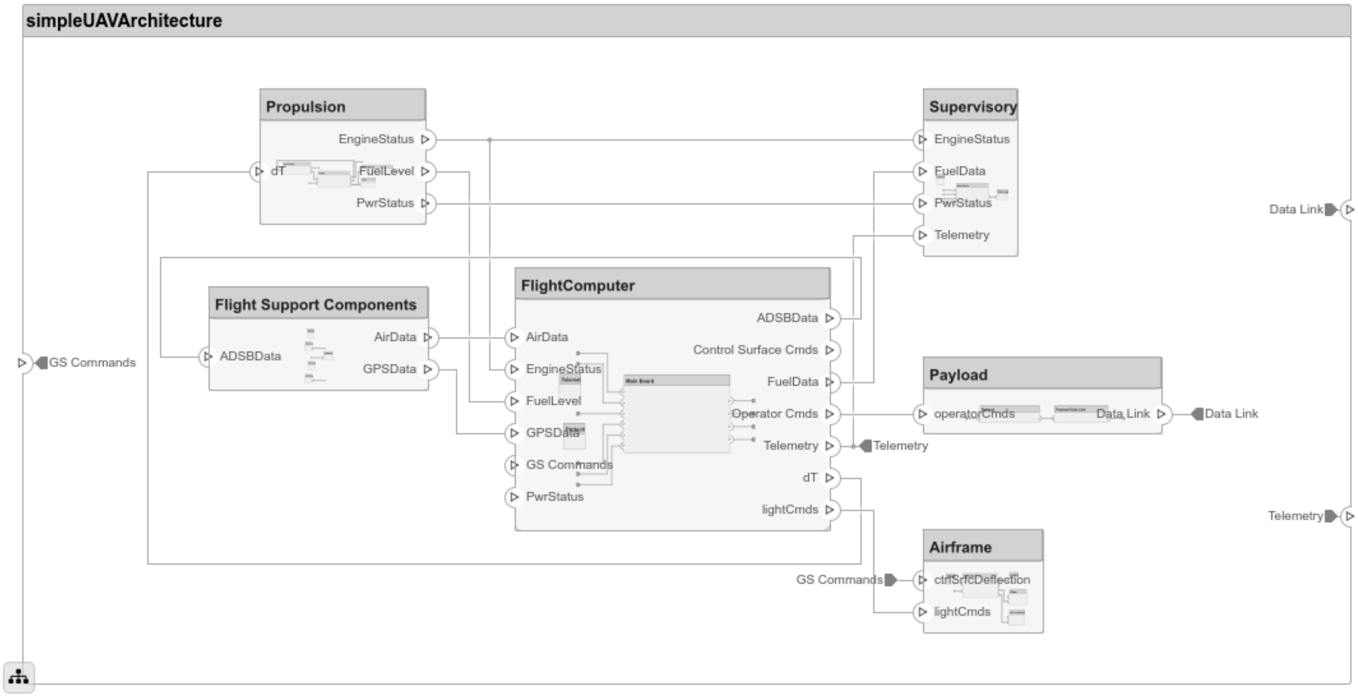
The systemcomposer.importModel(importModelName) API :

- 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);
```

See Also

`exportModel` | `importModel` | `updateLinksToReferenceRequirements`

More About

- “Import and Export Architecture Models” on page 1-50

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 i
# 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 lin
# 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
```

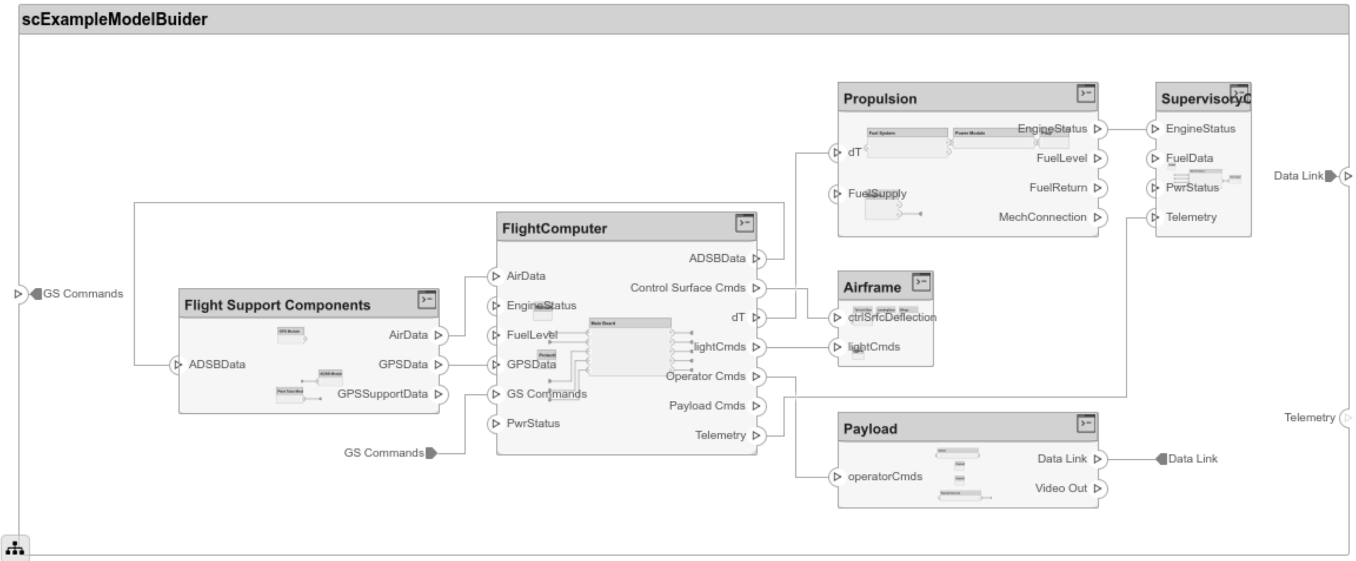
Step 3. Build Architecture Specifications

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);
```

See Also

exportModel | importModel | systemcomposer.io.ModelBuilder

More About

- “Import and Export Architecture Models” on page 1-50
- “Import and Export Architectures” on page 6-39

Simulating Mobile Robot with System Composer Workflow

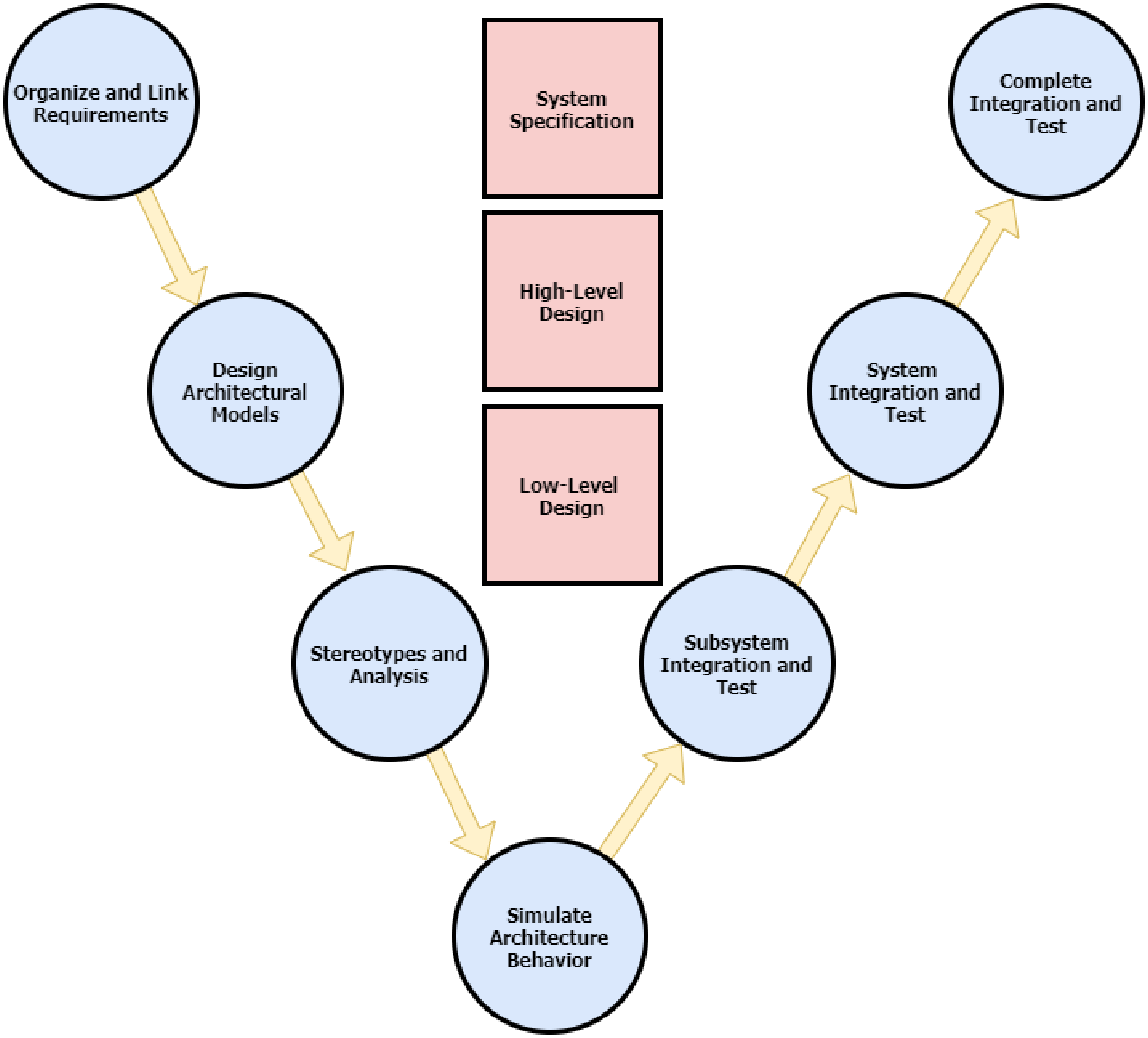
Overview

Along with other tools, System Composer™ can help you organize and link requirements, design and allocate architecture models, analyze the system, and implement the design in Simulink®. This example demonstrates the workflow for designing a system architecture using System Composer. This example follows the early phase development for a mobile robot:

- Setting up the requirements based on market research.
- Creating architecture models to help organize algorithms and hardware.
- Creating a Simulink model to simulate realistic behavior of the mobile robot.

This example describes a typical workflow for developing an autonomous mobile robot and conducting system analysis to ensure that the life expectancy of the durable components in the robot meets the customer-specified mean time before repair.

This example follows early workflow steps from “Model-Based Design with Simulink”.


Organize and Link Requirements
Design Architectural Models
Stereotypes and Analysis
Simulate Architecture Behavior
Subsystem Integration and Test
System Integration and Test
Complete Integration and Test
System Specification
High-Level Design
Low-Level Design

Organize and Link Requirements

The first step in model-based design is to set up requirements. In this example, there are three sets of requirements.

1. Stakeholder needs - a set of end-user needs.
2. System requirements - an organized set of requirements that are linked closely with the system-level design.
3. Implementation requirements - a detailed set of requirements that specify the model's subsystems.

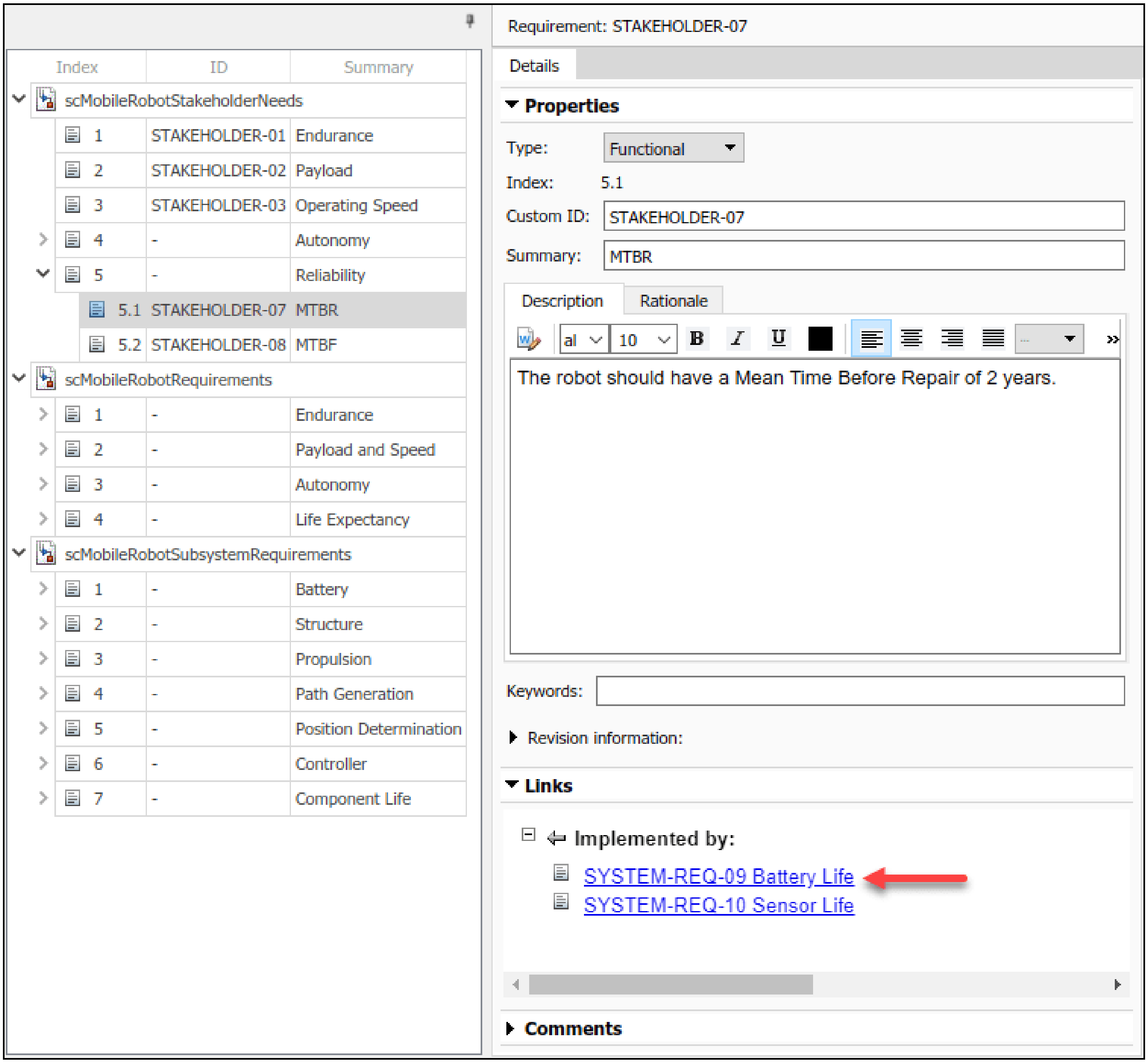
By linking one requirement set to another, each high-level requirement can be traced all the way to implementation. For more information on requirement links, see “Requirement Links” (Simulink Requirements).

Link Stakeholder Requirements to Technical Requirements

Navigate to the example folder. To open `scMobileRobotStakeholderNeeds.slreqx`, `scMobileRobotRequirements.slreqx`, and `scMobileRobotSubsystemRequirements.slreqx`, double-click each file or run this code in the MATLAB® Command Window:

```
load_system('scMobileRobotHardwareArchitecture.slx') % Load systems in memory to view requirement
load_system('scMobileRobotFunctionalArchitecture.slx')
open('scMobileRobotStakeholderNeeds.slreqx')
open('scMobileRobotRequirements.slreqx')
open('scMobileRobotSubsystemRequirements.slreqx')
```

Link stakeholder needs to derived requirements to keep track of high-level goals. Some links are already defined in this example, like the implementation link from `STAKEHOLDER-07` to `SYSTEM-REQ-09`. This information is saved in the file `scMobileRobotStakeholderNeeds.slmx`.



Index	ID	Summary
scMobileRobotStakeholderNeeds		
1	STAKEHOLDER-01	Endurance
2	STAKEHOLDER-02	Payload
3	STAKEHOLDER-03	Operating Speed
4	-	Autonomy
5	-	Reliability
5.1	STAKEHOLDER-07	MTBR
5.2	STAKEHOLDER-08	MTBF
scMobileRobotRequirements		
1	-	Endurance
2	-	Payload and Speed
3	-	Autonomy
4	-	Life Expectancy
scMobileRobotSubsystemRequirements		
1	-	Battery
2	-	Structure
3	-	Propulsion
4	-	Path Generation
5	-	Position Determination
6	-	Controller
7	-	Component Life

Requirement: STAKEHOLDER-07

Details

Properties

Type: Functional

Index: 5.1

Custom ID: STAKEHOLDER-07

Summary: MTBR

Description | Rationale

The robot should have a Mean Time Before Repair of 2 years.

Keywords:

Revision information:

Links

Implemented by:

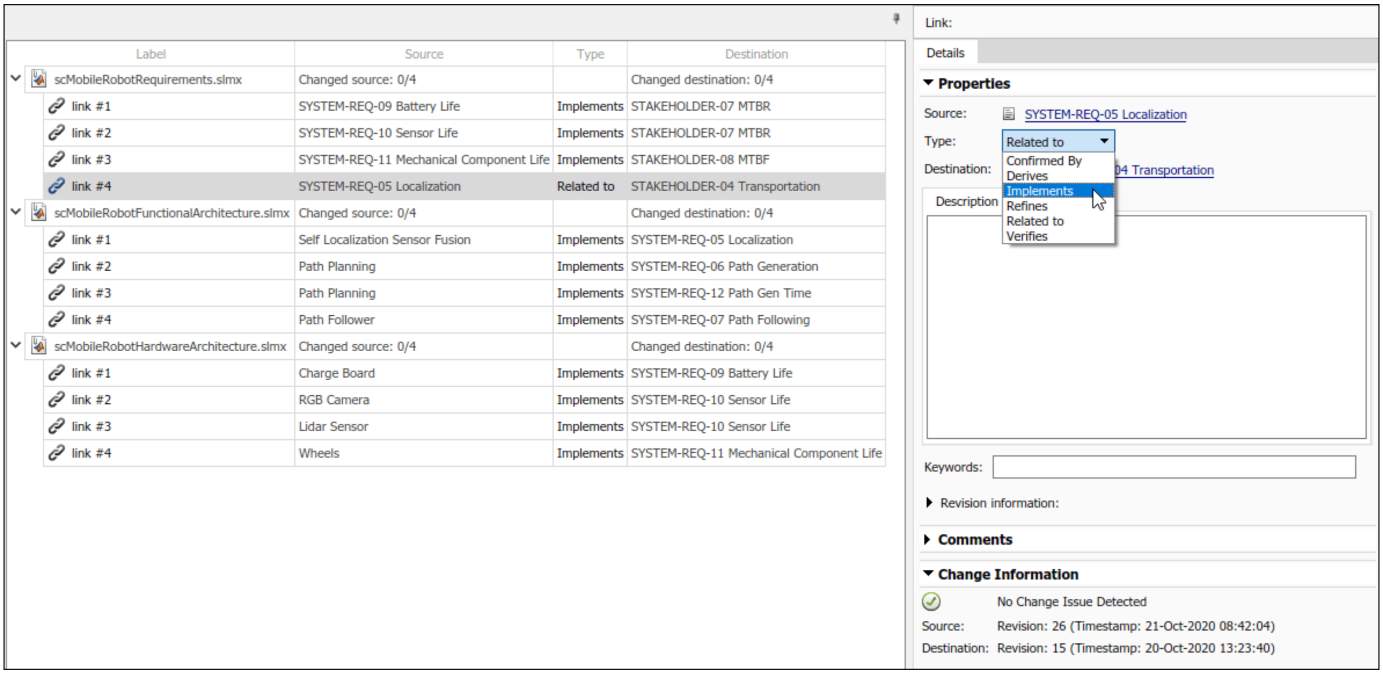
SYSTEM-REQ-09 Battery Life

SYSTEM-REQ-10 Sensor Life

Comments

You can create another link for the `Autonomy` requirement. The stakeholder's need to relocate an object with a specified tolerance `STAKEHOLDER-04` will be implemented by the system requirement `SYSTEM-REQ-05`. The robot must be able to determine its current position with a specified tolerance. Right-click `SYSTEM-REQ-05` and select `'Select for Linking with Requirement'`. Then, right-click on `STAKEHOLDER-04` and select `Create a link from SYSTEM-REQ-05 to STAKEHOLDER-04`.

More details on the links can be found under **View > Links**. Change the type of link to `Implements` since `STAKEHOLDER-04` is implemented by `SYSTEM-REQ-05`. For more information about link types, refer to “Define Custom Requirement and Link Types” (Simulink Requirements).



Label	Source	Type	Destination
scMobileRobotRequirements.slmx	Changed source: 0/4		Changed destination: 0/4
link #1	SYSTEM-REQ-09 Battery Life	Implements	STAKEHOLDER-07 MTBR
link #2	SYSTEM-REQ-10 Sensor Life	Implements	STAKEHOLDER-07 MTBR
link #3	SYSTEM-REQ-11 Mechanical Component Life	Implements	STAKEHOLDER-08 MTBF
link #4	SYSTEM-REQ-05 Localization	Related to	STAKEHOLDER-04 Transportation
scMobileRobotFunctionalArchitecture.slmx	Changed source: 0/4		Changed destination: 0/4
link #1	Self Localization Sensor Fusion	Implements	SYSTEM-REQ-05 Localization
link #2	Path Planning	Implements	SYSTEM-REQ-06 Path Generation
link #3	Path Planning	Implements	SYSTEM-REQ-12 Path Gen Time
link #4	Path Follower	Implements	SYSTEM-REQ-07 Path Following
scMobileRobotHardwareArchitecture.slmx	Changed source: 0/4		Changed destination: 0/4
link #1	Charge Board	Implements	SYSTEM-REQ-09 Battery Life
link #2	RGB Camera	Implements	SYSTEM-REQ-10 Sensor Life
link #3	Lidar Sensor	Implements	SYSTEM-REQ-10 Sensor Life
link #4	Wheels	Implements	SYSTEM-REQ-11 Mechanical Component Life

Link:

Details

Properties

Source: SYSTEM-REQ-05 Localization

Type: Related to

Destination: 04 Transportation

Description

Keywords:

Revision information:

Comments

Change Information

No Change Issue Detected

Source: Revision: 26 (Timestamp: 21-Oct-2020 08:42:04)

Destination: Revision: 15 (Timestamp: 20-Oct-2020 13:23:40)

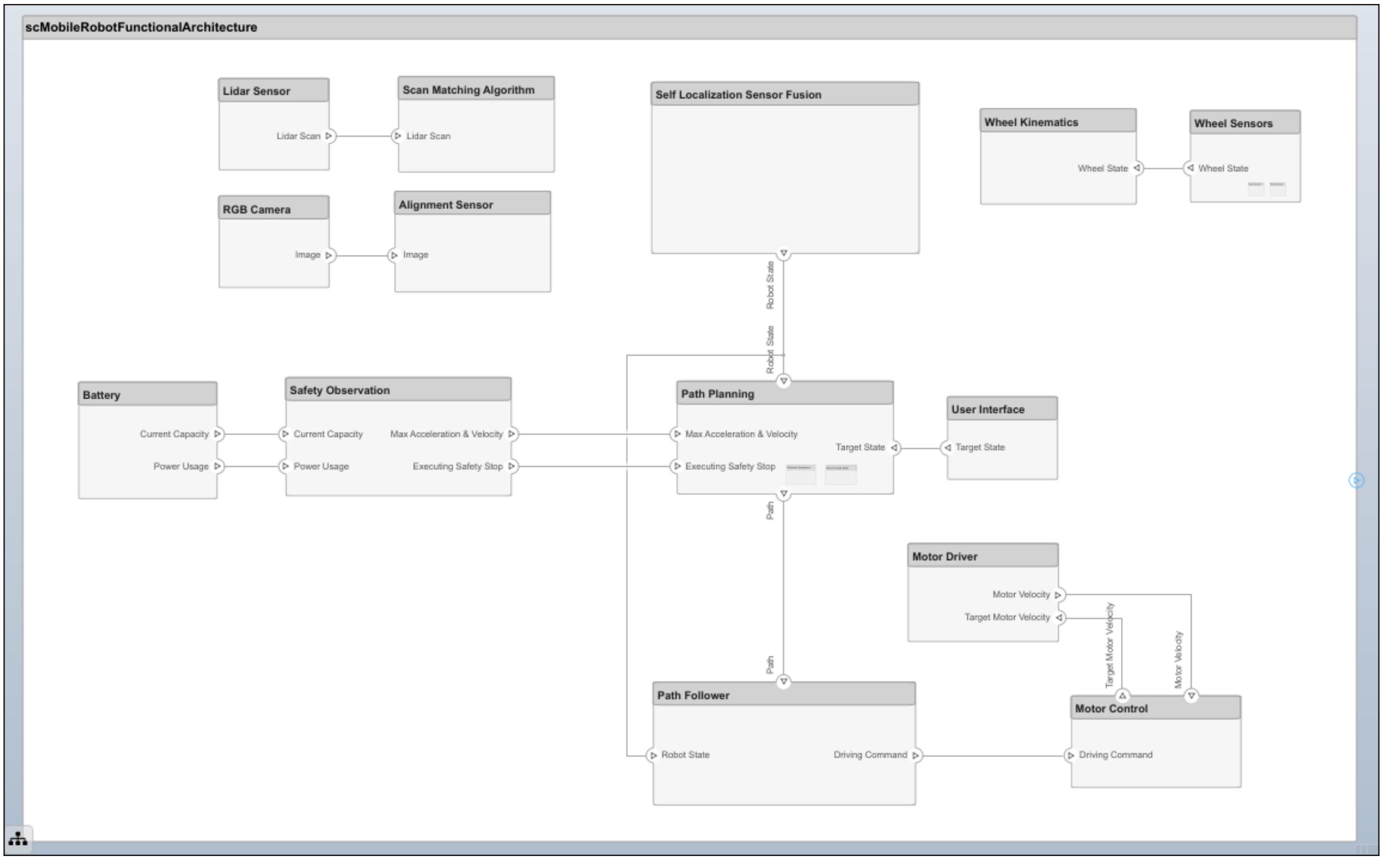
Design Architectural Models

Architecture models describe a system at different levels of abstraction. This example presents three architectures:

1. *Functional architecture* describes high-level functions.
2. *Hardware architecture* describes the physical hardware or platform needed for the robot.
3. *Logical architecture* describes data exchange.

To open the functional architecture, double-click the file or enter the model name into the MATLAB Command Window.

```
scMobileRobotFunctionalArchitecture
```

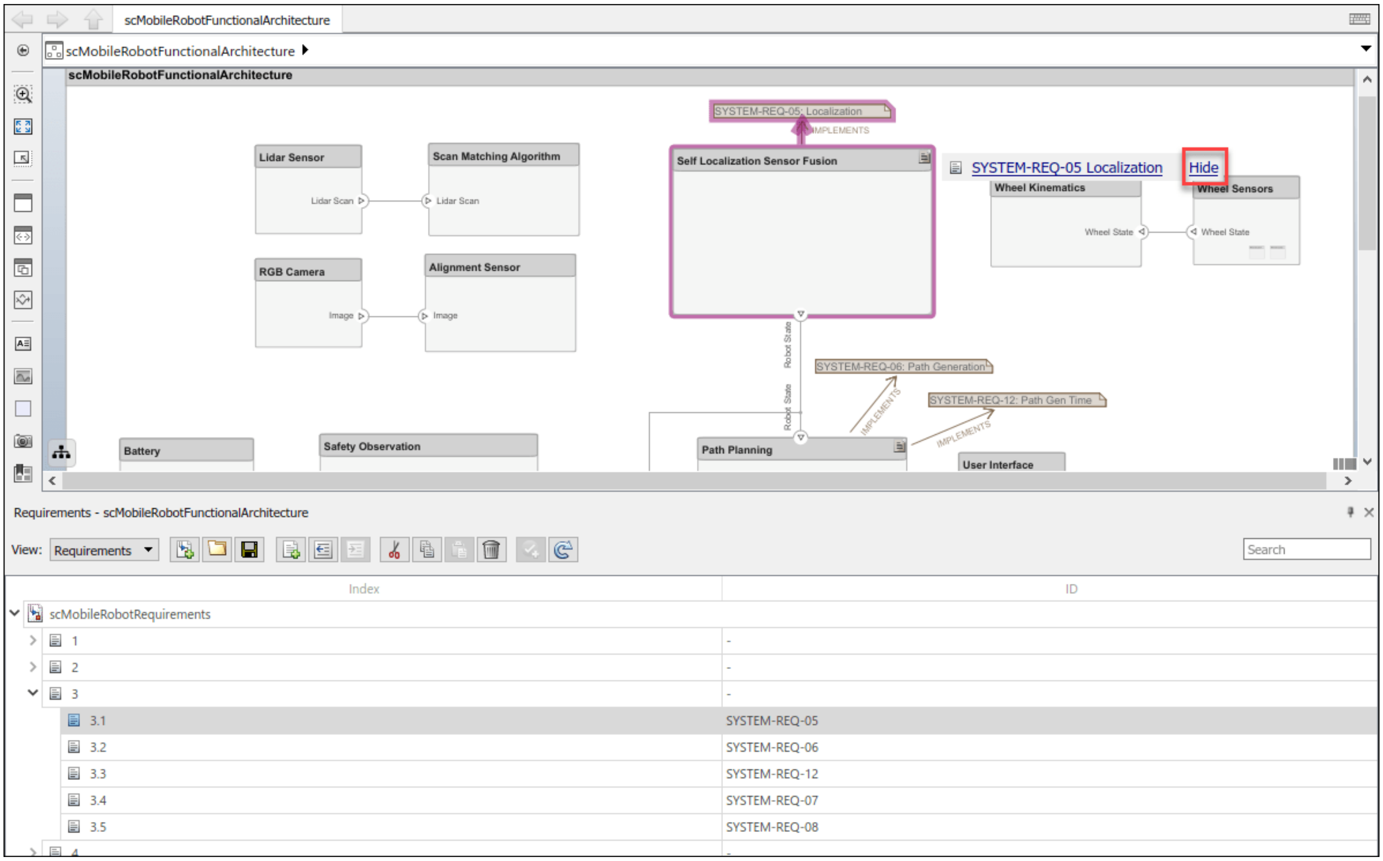


The functional architecture describes functional dependencies: controlling a mobile robot autonomously, localization, path planning, and path following.

Link Requirements to Architecture Models

Requirement traceability involves linking technical requirements to architecture models, thereby allowing the connection between an early requirements phase and system-level design. You can more easily track whether a requirement is met by connecting components back to stakeholder needs. To view requirements, open the Requirements Manager by accessing **Apps > Requirements Manager**.

The `'Self Localization Sensor Fusion'` component in functional architecture implements the `'Localization'` requirement `SYSTEM-REQ-05`. To show or hide linked requirements, click the linked icon on the top-right corner of a component.


scMobileRobotFunctionalArchitecture
scMobileRobotFunctionalArchitecture
scMobileRobotFunctionalArchitecture
SYSTEM-REQ-05: Localization
IMPLEMENTS
Lidar Sensor
Lidar Scan
Scan Matching Algorithm
Lidar Scan
Self Localization Sensor Fusion
SYSTEM-REQ-05 Localization
Hide
Wheel Kinematics
Wheel State
Wheel Sensors
Wheel State
RGB Camera
Image
Alignment Sensor
Image
Robot State
SYSTEM-REQ-06: Path Generation
SYSTEM-REQ-12: Path Gen Time
IMPLEMENTS
IMPLEMENTS
Robot State
Battery
Safety Observation
Path Planning
User Interface
Requirements - scMobileRobotFunctionalArchitecture
View:
Requirements
Search

Index	ID
scMobileRobotRequirements	
1	-
2	-
3	-
3.1	SYSTEM-REQ-05
3.2	SYSTEM-REQ-06
3.3	SYSTEM-REQ-12
3.4	SYSTEM-REQ-07
3.5	SYSTEM-REQ-08
4	-

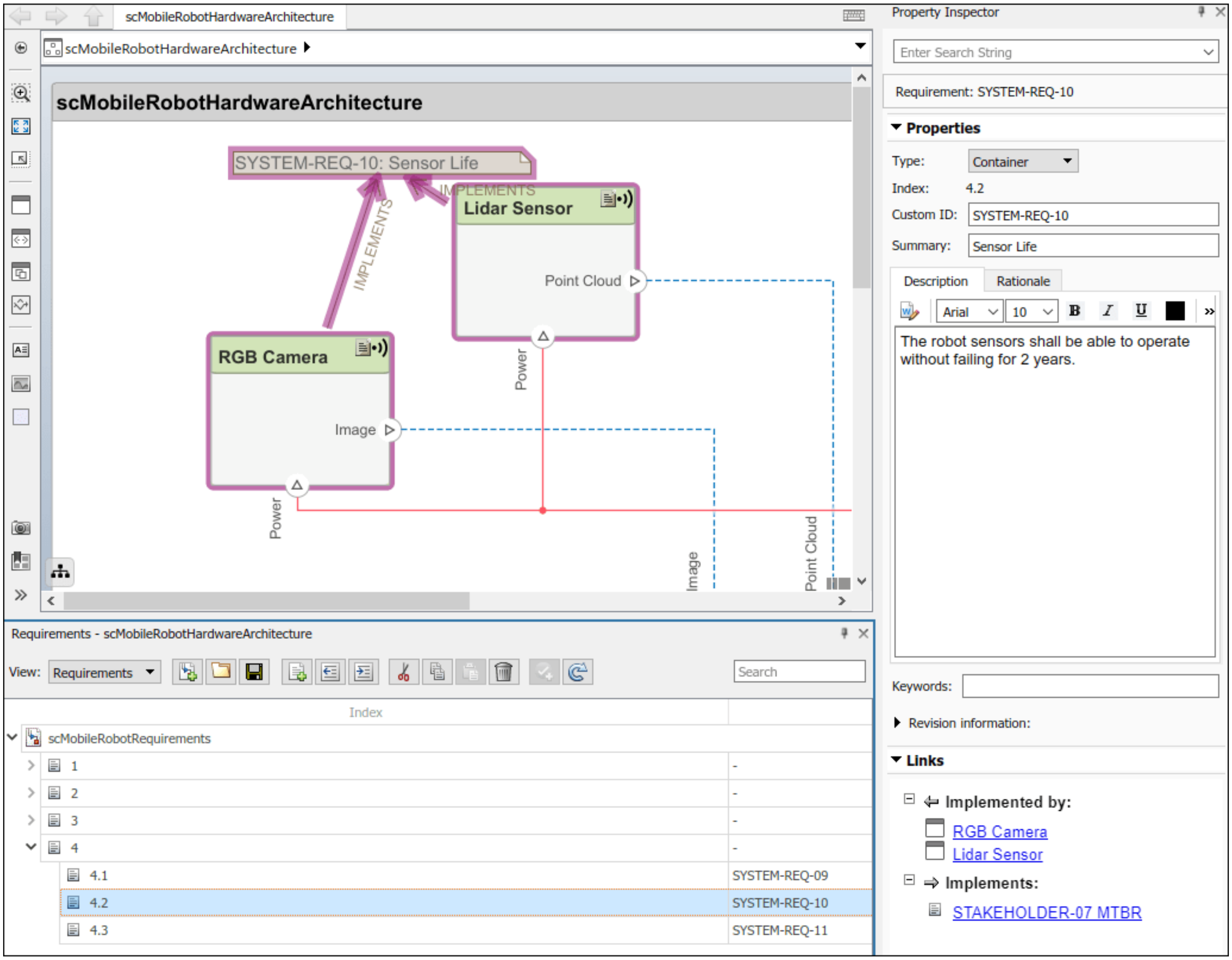
You can add links by dragging requirements to a component.

Open the hardware architecture model. To open the model, double-click the file or enter the model name in the MATLAB Command Window.

```
scMobileRobotHardwareArchitecture
```

The hardware architecture model describes the hardware components — the sensor, actuators, and embedded processor — and their connections. The colors and icons indicate the stereotypes used for each element.

You can view the requirements linked to the hardware architecture model in the Requirement Manager.



Requirement: SYSTEM-REQ-10

Properties

Type: Container
Index: 4.2
Custom ID: SYSTEM-REQ-10
Summary: Sensor Life

Description | Rationale

The robot sensors shall be able to operate without failing for 2 years.

Keywords:

Revision information:

Links

Implemented by:
- RGB Camera
- Lidar Sensor

Implements:
- STAKEHOLDER-07 MTBR

Index	
scMobileRobotRequirements	
1	-
2	-
3	-
4	-
4.1	SYSTEM-REQ-09
4.2	SYSTEM-REQ-10
4.3	SYSTEM-REQ-11

Only requirements related to `'Life Expectancy'` are shown. Link other requirements by dragging and dropping from the Requirements Editor. For more information about requirements, see “Link and Trace Requirements” on page 6-25.

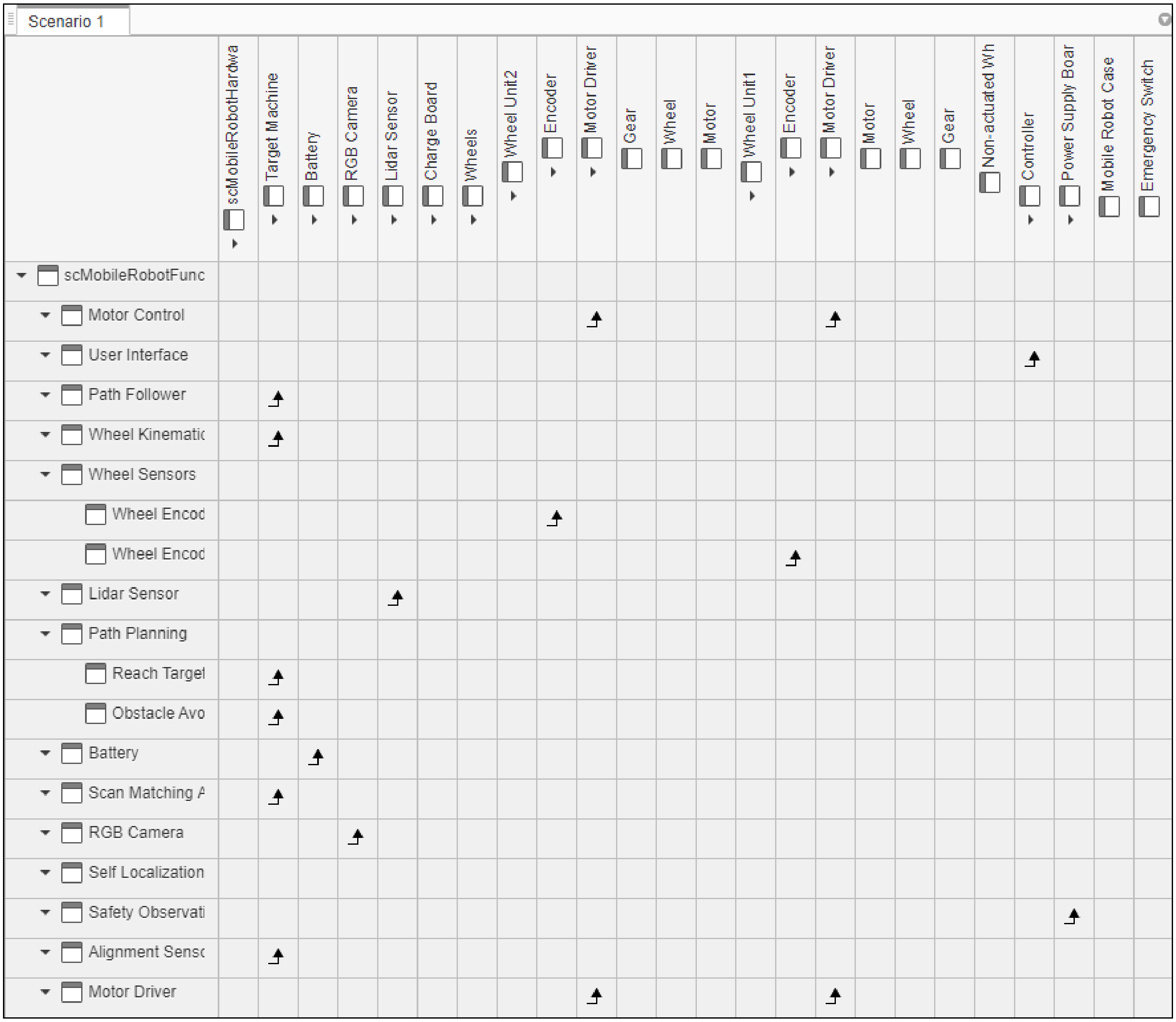
Link Functional to Hardware Architecture Using Allocations

You can allocate functional architecture to hardware architecture using the Allocation Editor.

To open the Allocation Editor, click **Modeling > Allocation Editor**, or enter the following into the Command Window.

```
systemcomposer.allocation.editor
```

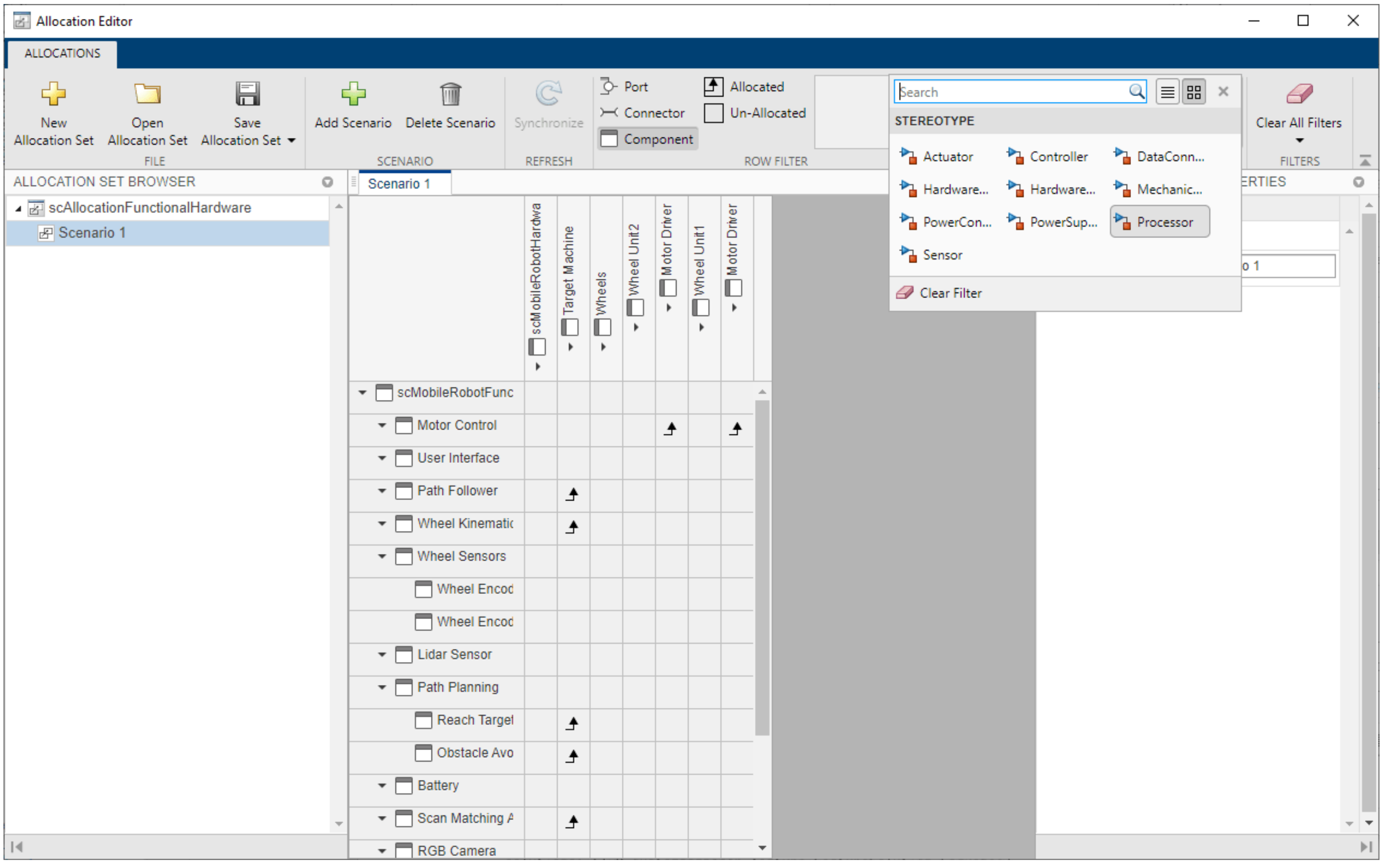
Load the allocation set: `scAllocationFunctionalHardware.mldatx` in the editor. Click on `'Scenario 1'`. Click `'Component'` in the **Row Filter** and **Column Filter** sections to filter rows and columns by components. The Allocation Editor allows you to link different architecture models to establish traceability for your project. Components of the functional architecture are allocated to components of the hardware architecture.



Scenario 1

	scMobileRobotHardwa	Target Machine	Battery	RGB Camera	Lidar Sensor	Charge Board	Wheels	Wheel Unit2	Encoder	Motor Driver	Gear	Wheel	Motor	Wheel Unit1	Encoder	Motor Driver	Motor	Wheel	Gear	Non-actuated Wh	Controller	Power Supply Boar	Mobile Robot Case	Emergency Switch
scMobileRobotFunc																								
Motor Control										↑						↑								
User Interface																					↑			
Path Follower		↑																						
Wheel Kinematic		↑																						
Wheel Sensors																								
Wheel Encod									↑															
Wheel Encod															↑									
Lidar Sensor					↑																			
Path Planning																								
Reach Target		↑																						
Obstacle Avo		↑																						
Battery			↑																					
Scan Matching A		↑																						
RGB Camera				↑																				
Self Localization																								
Safety Observati																						↑		
Alignment Sensc		↑																						
Motor Driver										↑						↑								

Click '`Processor`' on the **Column Filter** to filter the allocations further. Since stereotypes are applied only to the hardware architecture in this example, the stereotype filter displays in the **Column Filter**.



	scMobileRobotHardwa	Target Machine	Wheels	Wheel Unit2	Motor Driver	Wheel Unit1	Motor Driver
scMobileRobotFunc							
Motor Control					↑		↑
User Interface							
Path Follower		↑					
Wheel Kinematic		↑					
Wheel Sensors							
Wheel Encod							
Wheel Encod							
Lidar Sensor							
Path Planning							
Reach Target		↑					
Obstacle Avo		↑					
Battery							
Scan Matching A		↑					
RGB Camera							

The autonomy of a vehicle is mostly handled by a target machine, which is an embedded computer responsible for processing sensor readings to calculate control inputs. Therefore, many functional components like 'Path Follower', 'Wheel Kinematics', and 'Scan Matching Algorithms' are allocated to the 'Target Machine'.

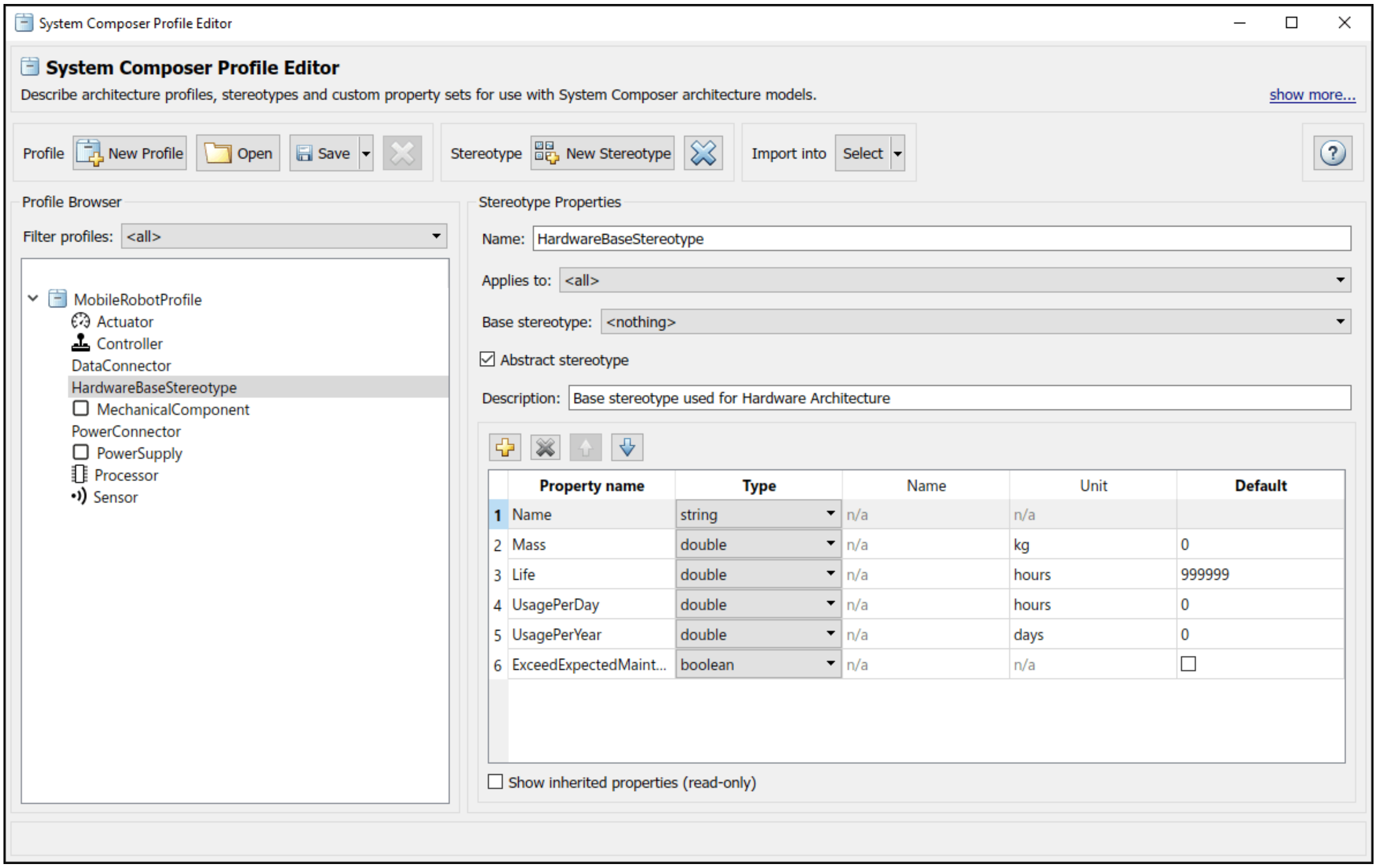
You can add allocations for ports and connectors as well. Double-click the intersections of the table to allocate or deallocate two elements.For more information on allocation, see “Allocate Architectures in a Tire Pressure Monitoring System” on page 6-5.

Stereotypes and Analysis

Stereotypes add an additional layer of metadata to components, ports, and connectors. The hardware architecture provides a basis to understand the applied stereotypes. To create a profile, see “Define Profiles and Stereotypes” on page 4-2.

In this example, the HardwareBaseStereotype is defined as 'Abstract Stereotype' and is extended to connector and component stereotypes. For example, a DataConnector stereotype is a connector stereotype that inherits the HardwareBaseStereotype. Other than properties like name and mass, the DataConnector stereotype has an additional property, TypeOfConnection, that describes which of the three connection types — RS232, Ethernet, or USB — it uses. For more information on setting up profiles, see “Use Stereotypes and Profiles” on page 4-10.

To focus on expected time before first maintenance, define properties such as 'UsagePerDay', 'UsagePerYear', and 'Life'. Setting these properties allows you to analyze each hardware component to make sure the mobile robot will last until first expected year of maintenance. To open the Profile Editor, go to **Modeling > Profiles > Edit**.



System Composer Profile Editor

Describe architecture profiles, stereotypes and custom property sets for use with System Composer architecture models. show more...

Filter profiles: <all>

- MobileRobotProfile
  - Actuator
  - Controller
  - DataConnector
  - HardwareBaseStereotype
  - MechanicalComponent
  - PowerConnector
  - PowerSupply
  - Processor
  - Sensor

Name: HardwareBaseStereotype

Applies to: <all>

Base stereotype: <nothing>

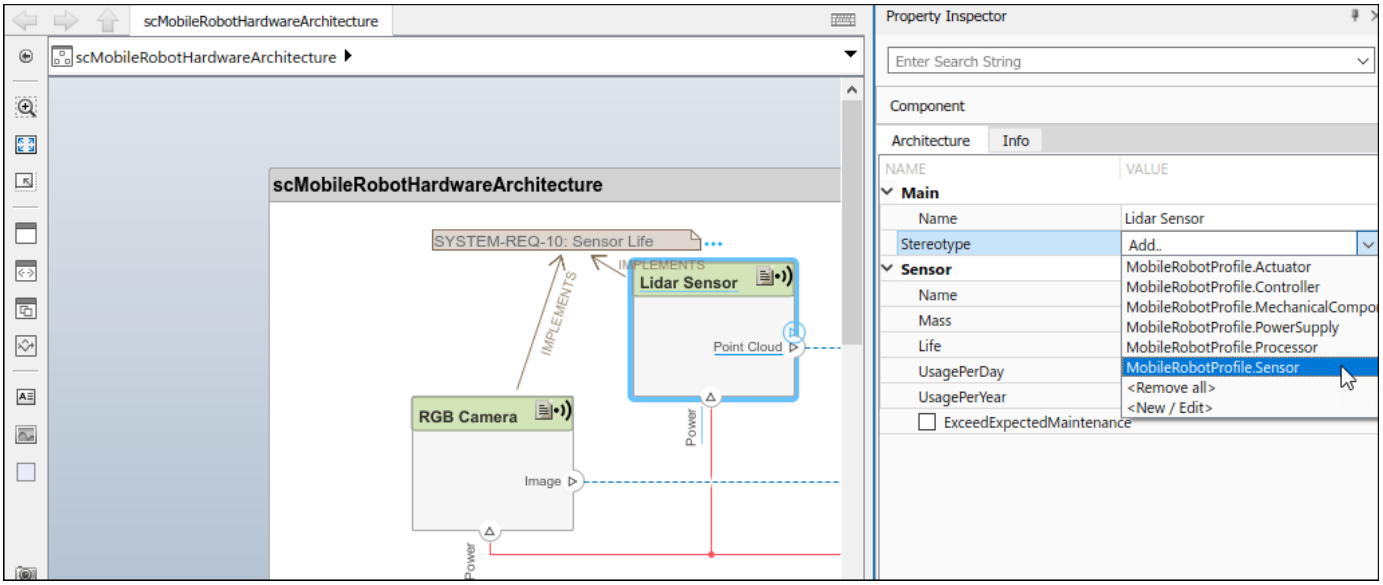
☑ Abstract stereotype

Description: Base stereotype used for Hardware Architecture

	Property name	Type	Name	Unit	Default
1	Name	string	n/a	n/a	
2	Mass	double	n/a	kg	0
3	Life	double	n/a	hours	999999
4	UsagePerDay	double	n/a	hours	0
5	UsagePerYear	double	n/a	days	0
6	ExceedExpectedMaint...	boolean	n/a	n/a	□

☐ Show inherited properties (read-only)

Once you define stereotypes in the Profile Editor, you can apply them to components and connectors. Apply stereotypes using the Property Inspector. To open the Property Inspector, go to **Modeling > Design > Property Inspector**.


scMobileRobotHardwareArchitecture
scMobileRobotHardwareArchitecture
scMobileRobotHardwareArchitecture
SYSTEM-REQ-10: Sensor Life
IMPLEMENTS
IMPLEMENTS
Lidar Sensor
Point Cloud
Power
RGB Camera
Image
Power
Property Inspector
Enter Search String
Architecture
Info

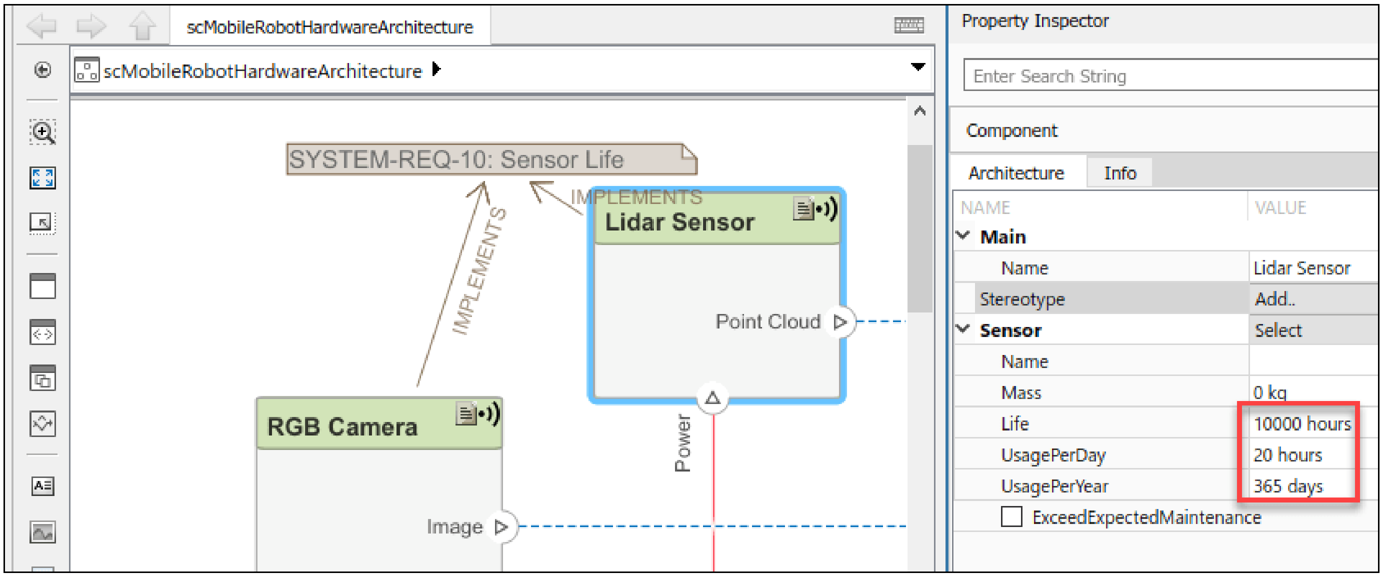
Component

NAME	VALUE
Main	
Name	Lidar Sensor
Stereotype	Add..
Sensor	
Name	
Mass	
Life	
UsagePerDay	
UsagePerYear	
□ ExceedExpectedMaintenance	

- MobileRobotProfile.Actuator
- MobileRobotProfile.Controller
- MobileRobotProfile.MechanicalCompo
- MobileRobotProfile.PowerSupply
- MobileRobotProfile.Processor
- MobileRobotProfile.Sensor
- <Remove all>
- <New / Edit>

To add stereotypes to elements, select the element in the diagram. In the Property Inspector, go to **Main > Stereotype**. Multiple stereotypes can be applied to the same element. Apply the `MobileRobotProfile.Sensor` stereotype to the `Lidar Sensor` component to add relevant properties.

Some components are in use for longer periods of time than others. The `Lidar Sensor` component is used for obstacle avoidance in this scenario so it is always in use except when it is charging. The `RGB Camera` only aligns the robot to the charging station, so it is in use for a shorter period per day. You can change values for the `'UsagePerDay'`, `'UsagePerYear'`, and `'Life'` properties to determine the expected maintenance time for components that are each used with different frequency.


scMobileRobotHardwareArchitecture
scMobileRobotHardwareArchitecture ▶
SYSTEM-REQ-10: Sensor Life
IMPLEMENTS
IMPLEMENTS
Lidar Sensor
Point Cloud
Power
RGB Camera
Image

Property Inspector

Enter Search String

Component

Architecture | Info

NAME	VALUE
Main	
Name	Lidar Sensor
Stereotype	Add..
Sensor	Select
Name	
Mass	0 kg
Life	10000 hours
UsagePerDay	20 hours
UsagePerYear	365 days
□ ExceedExpectedMaintenance	

The property `'ExceedExpectedMaintenance'` is set to `'False'` by default. This property will update when you run your analysis.

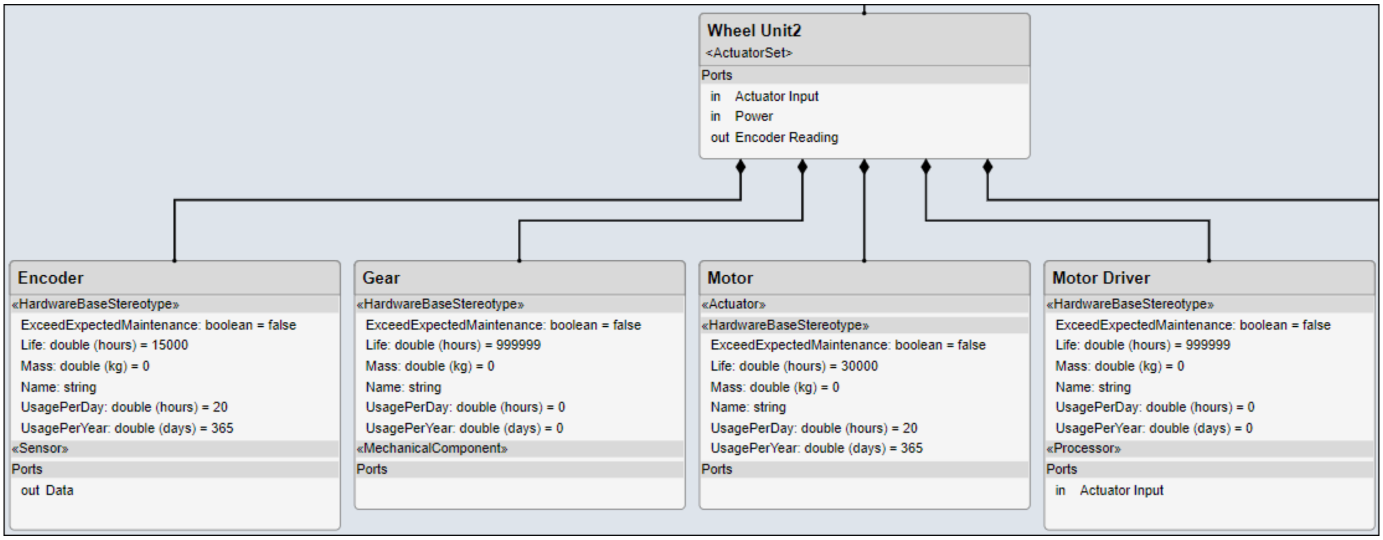
Architecture Views Gallery

Use the Architecture Views Gallery to review changes you make in the architecture model. Architecture views allow you to create filtered views and thereby focus on few elements of the model, which enables you to navigate a complex model more easily. For example, an electrical engineer might be interested only in the electrical components of the hardware architecture. The engineer could apply a filter to show only components with electrical stereotypes.

In this example, you would apply the filter to view components with regard to the `'Life Expectancy'` requirement.

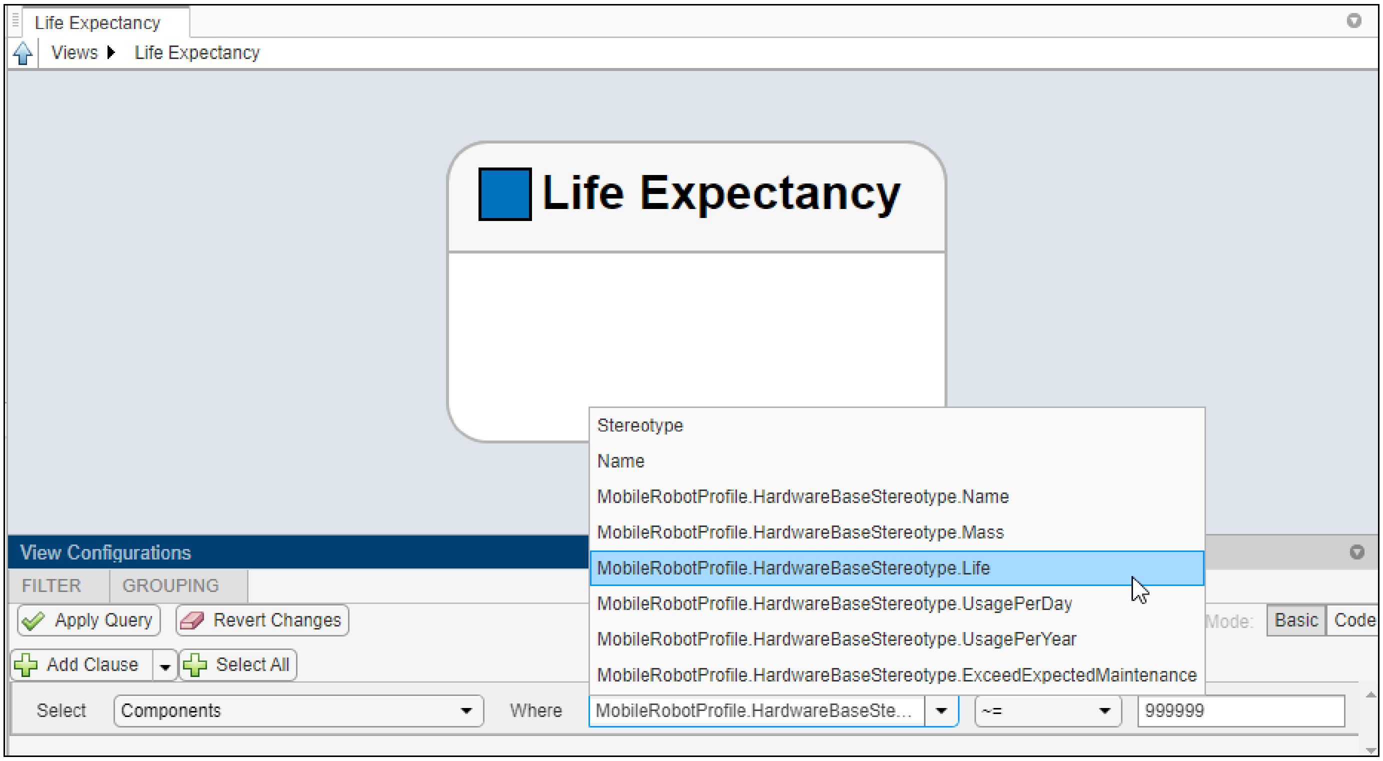
To open the Architecture Views Gallery, go to **Views > Architecture Views**.

Click **New View**, then click **Select All** to select all components for view. Click **Apply Query**. Select the **Hierarchy Diagram** view. The hierarchy of the components is flattened to show all subcomponents in one view.


Wheel Unit2
<ActuatorSet>
Ports
in Actuator Input
in Power
out Encoder Reading
Encoder
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 15000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 20
UsagePerYear: double (days) = 365
«Sensor»
Ports
out Data
Gear
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 999999
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 0
UsagePerYear: double (days) = 0
«MechanicalComponent»
Ports
Motor
«Actuator»
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 30000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 20
UsagePerYear: double (days) = 365
Ports
Motor Driver
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 999999
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 0
UsagePerYear: double (days) = 0
«Processor»
Ports
in Actuator Input

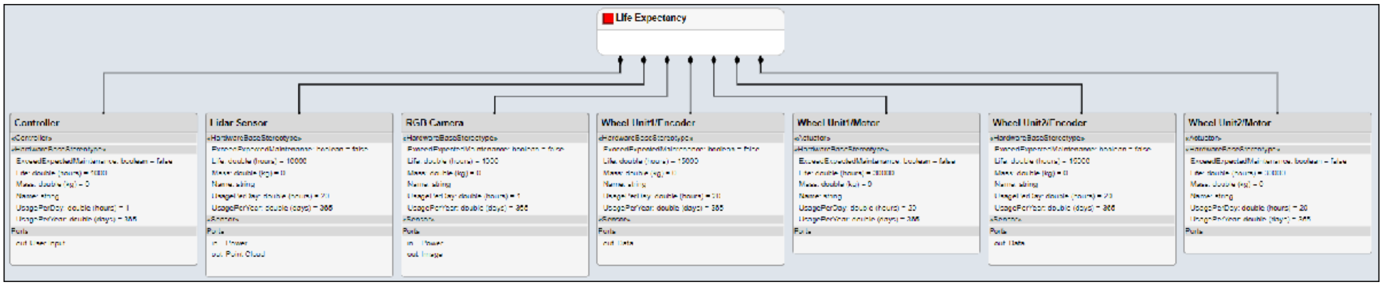
You can add filter logic to view relevant components for the `'Life Expectancy'` requirement. Click **New View**, then click **Add Clause**.

Note that the default value for the `MobileRobotProfile.HardwareBaseStereotype.Life` stereotype is `999999`. Set the filter to show components with `MobileRobotProfile.HardwareBaseStereotype.Life` not equal to `999999`, which will indicate whether or not the stereotype was set.


Life Expectancy
Views ▸ Life Expectancy
Life Expectancy
View Configurations
FILTER
GROUPING
Apply Query
Revert Changes
Add Clause
Select All
Mode: Basic Code
Stereotype
Name
MobileRobotProfile.HardwareBaseStereotype.Name
MobileRobotProfile.HardwareBaseStereotype.Mass
MobileRobotProfile.HardwareBaseStereotype.Life
MobileRobotProfile.HardwareBaseStereotype.UsagePerDay
MobileRobotProfile.HardwareBaseStereotype.UsagePerYear
MobileRobotProfile.HardwareBaseStereotype.ExceedExpectedMaintenance

Select Components Where MobileRobotProfile.HardwareBaseSte... ~= 999999

Click **Apply Query**. Go to **Hierarchy Diagram** to view the components of interest.

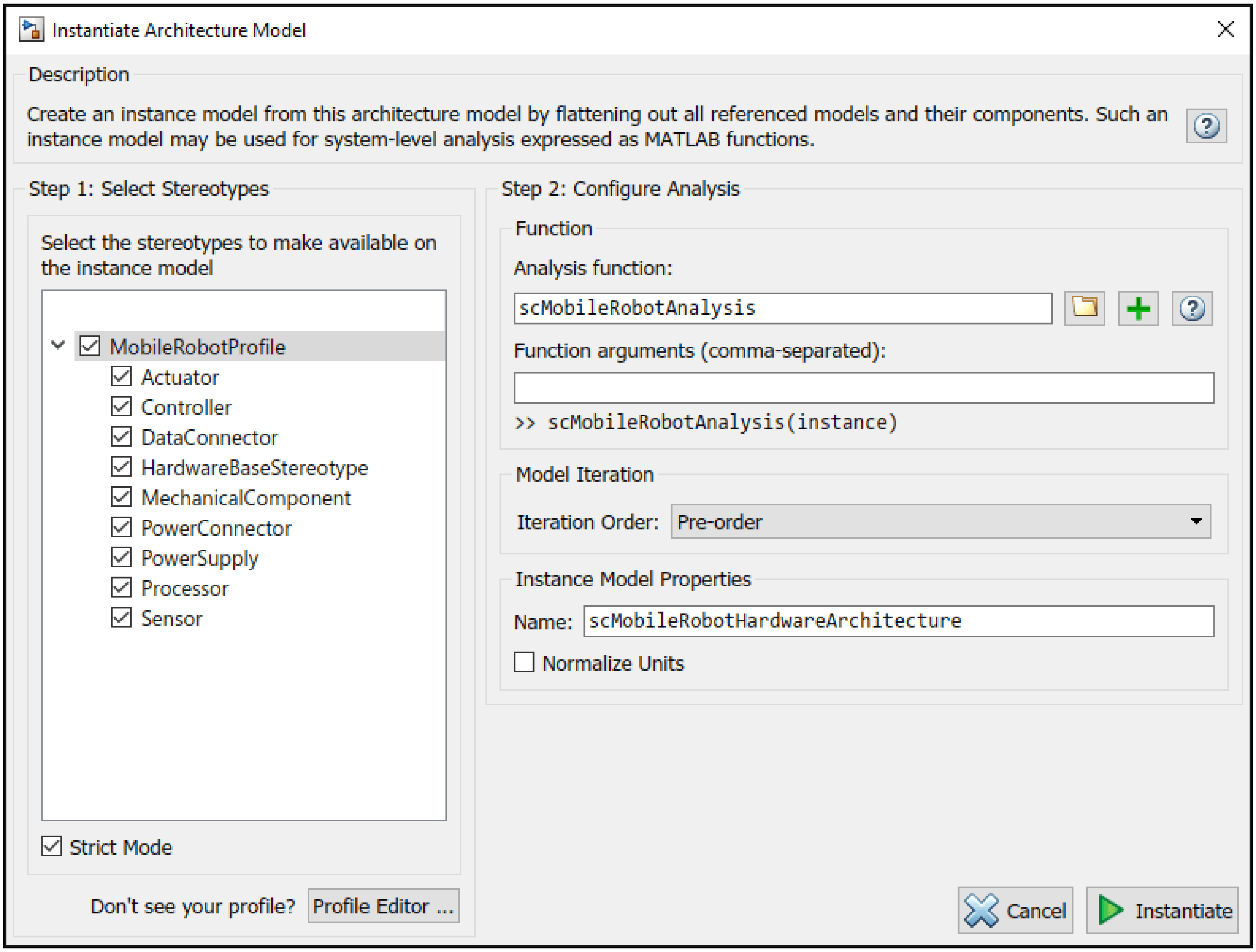

Life Expectancy
Controller
«Controller»
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 1000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 1
UsagePerYear: double (days) = 365
Ports
out User Input
Lidar Sensor
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 10000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 20
UsagePerYear: double (days) = 365
«Sensor»
Ports
in Power
out Point Cloud
RGB Camera
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 1000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 1
UsagePerYear: double (days) = 365
«Sensor»
Ports
in Power
out Image
Wheel Unit1/Encoder
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 15000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 20
UsagePerYear: double (days) = 365
«Sensor»
Ports
out Data
Wheel Unit1/Motor
«Actuator»
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 200000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 20
UsagePerYear: double (days) = 365
Ports
Wheel Unit2/Encoder
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 15000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 20
UsagePerYear: double (days) = 365
«Sensor»
Ports
out Data
Wheel Unit2/Motor
«Actuator»
«HardwareBaseStereotype»
ExceedExpectedMaintenance: boolean = false
Life: double (hours) = 10000
Mass: double (kg) = 0
Name: string
UsagePerDay: double (hours) = 20
UsagePerYear: double (days) = 365
Ports

For more information about architecture vews, see “Create Architecture Views Interactively” on page 1-37.

Model Analysis

Analyze the system to check if the components and connectors will last longer than the expected time before first maintenance. This value is set to 2 years in the analysis function. Open the Analysis Viewer under **Modeling > Views > Analysis Model**.

Select all stereotypes to make them available on the instance model. Choose `scMobileRobotAnalysis.m` as the analysis function. The iteration order determines in what order the component hierarchy is analyzed. However, since each component is analyzed separately, the order does not matter. Select the default `'Pre-order'`.



Instantiate Architecture Model

Description

Create an instance model from this architecture model by flattening out all referenced models and their components. Such an instance model may be used for system-level analysis expressed as MATLAB functions.

Step 1: Select Stereotypes

Select the stereotypes to make available on the instance model

- ☑ MobileRobotProfile
  - ☑ Actuator
  - ☑ Controller
  - ☑ DataConnector
  - ☑ HardwareBaseStereotype
  - ☑ MechanicalComponent
  - ☑ PowerConnector
  - ☑ PowerSupply
  - ☑ Processor
  - ☑ Sensor

☑ Strict Mode

Don't see your profile? Profile Editor ...

Step 2: Configure Analysis

Function

Analysis function: scMobileRobotAnalysis

Function arguments (comma-separated):

```
>> scMobileRobotAnalysis(instance)
```

Model Iteration

Iteration Order: Pre-order

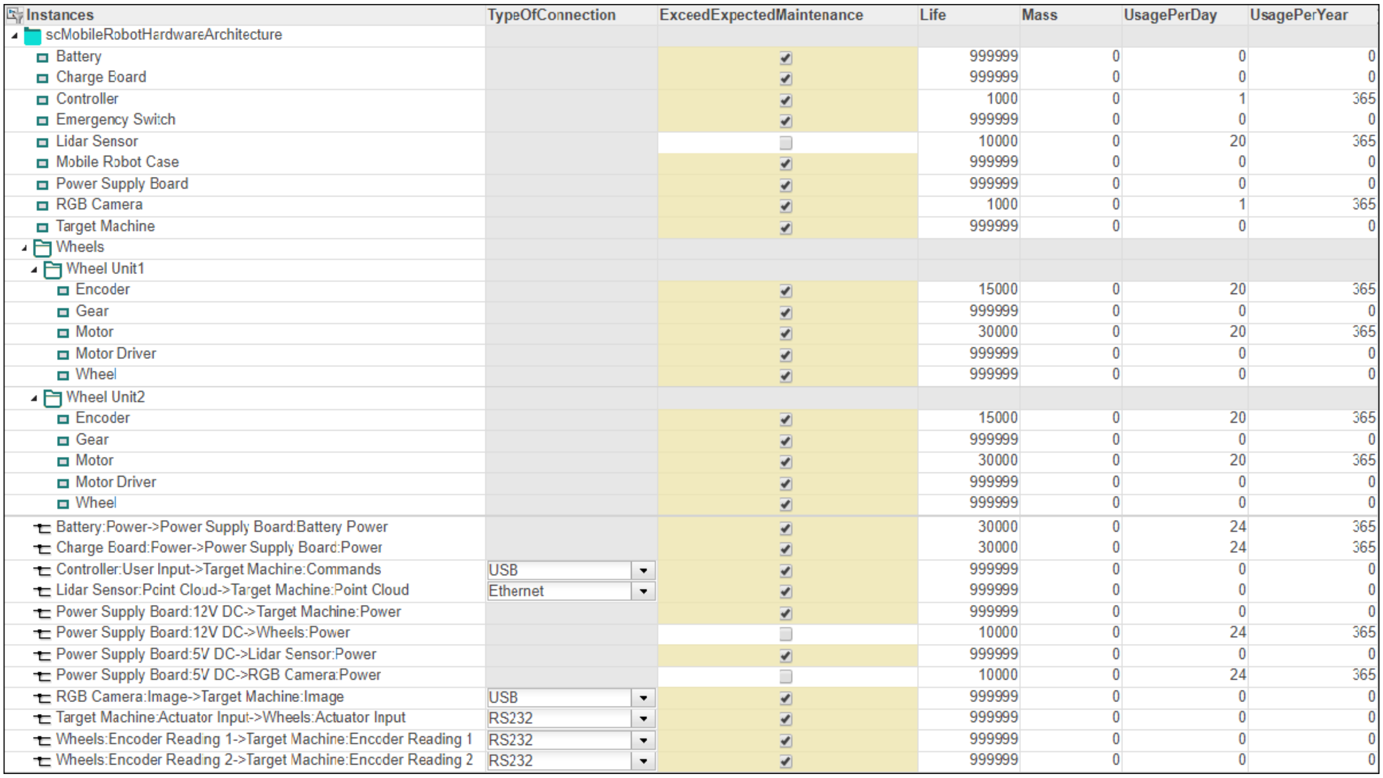
Instance Model Properties

Name: scMobileRobotHardwareArchitecture

□ Normalize Units

Click **Instantiate** to instantiate the model.

Relevant components and connectors with stereotypes are shown. Since all stereotypes are selected, all elements with stereotypes are shown in the instance model. Model analysis will calculate which components and connectors will last longer than the expected 2 years. Click **Analyze** to perform the calculation.



Instances	TypeOfConnection	ExceedExpectedMaintenance	Life	Mass	UsagePerDay	UsagePerYear
scMobileRobotHardwareArchitecture						
Battery		☑	999999	0	0	0
Charge Board		☑	999999	0	0	0
Controller		☑	1000	0	1	365
Emergency Switch		☑	999999	0	0	0
Lidar Sensor		□	10000	0	20	365
Mobile Robot Case		☑	999999	0	0	0
Power Supply Board		☑	999999	0	0	0
RGB Camera		☑	1000	0	1	365
Target Machine		☑	999999	0	0	0
Wheels						
Wheel Unit1						
Encoder		☑	15000	0	20	365
Gear		☑	999999	0	0	0
Motor		☑	30000	0	20	365
Motor Driver		☑	999999	0	0	0
Wheel		☑	999999	0	0	0
Wheel Unit2						
Encoder		☑	15000	0	20	365
Gear		☑	999999	0	0	0
Motor		☑	30000	0	20	365
Motor Driver		☑	999999	0	0	0
Wheel		☑	999999	0	0	0
Battery:Power->Power Supply Board:Battery Power		☑	30000	0	24	365
Charge Board:Power->Power Supply Board:Power		☑	30000	0	24	365
Controller:User Input->Target Machine:Commands	USB	☑	999999	0	0	0
Lidar Sensor:Point Cloud->Target Machine:Point Cloud	Ethernet	☑	999999	0	0	0
Power Supply Board:12V DC->Target Machine:Power		☑	999999	0	0	0
Power Supply Board:12V DC->Wheels:Power		□	10000	0	24	365
Power Supply Board:5V DC->Lidar Sensor:Power		☑	999999	0	0	0
Power Supply Board:5V DC->RGB Camera:Power		□	10000	0	24	365
RGB Camera:Image->Target Machine:Image	USB	☑	999999	0	0	0
Target Machine:Actuator Input->Wheels:Actuator Input	RS232	☑	999999	0	0	0
Wheels:Encoder Reading 1->Target Machine:Encoder Reading 1	RS232	☑	999999	0	0	0
Wheels:Encoder Reading 2->Target Machine:Encoder Reading 2	RS232	☑	999999	0	0	0

The analysis function `scMobileRobotAnalysis` calculates if `UsagePerDay*UsagePerYear*ExpectedYearsBeforeFirstMaintenance` will exceed `Life`, which is set to 2 years, for each component and connector. The unchecked boxes indicate that components and connectors will need maintenance within 2 years with the given specification.

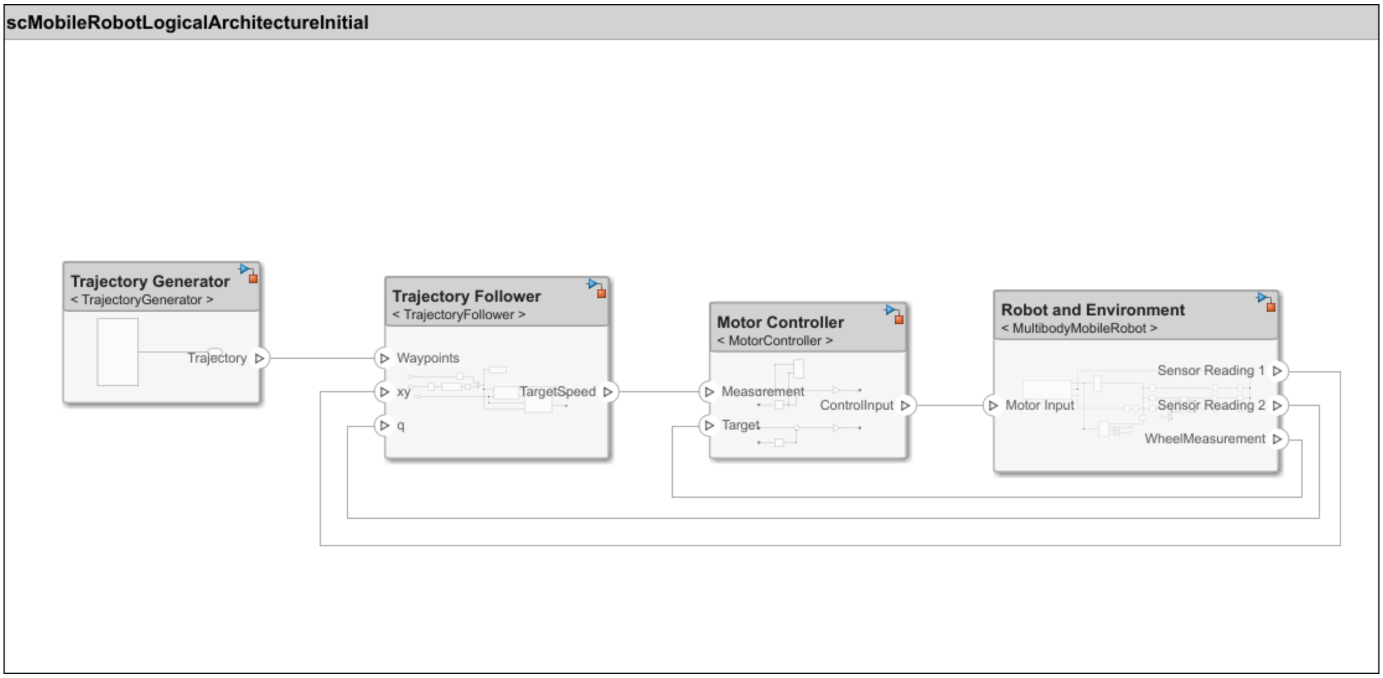
To refresh the instance model in the Analysis Viewer, select `Overwrite`, then click **Refresh**. This action will retrieve the values back from the source model, in this case, the hardware architecture model. Since `ExceedExpectedMaintenance` was the only property changed, it will be changed to its default value. Conversely, clicking **Update** will change the property values in the hardware architecture source according to the instance model. For more on analysis, see “Analyze Architecture” on page 6-10.

Simulate Architecture Behavior

You can inspect the logical architecture and link to Simulink behavior models to run the simulation.

To open the logical architecture, double-click the file, or enter the file name in the MATLAB Command Window.

```
scMobileRobotLogicalArchitectureInitial
```



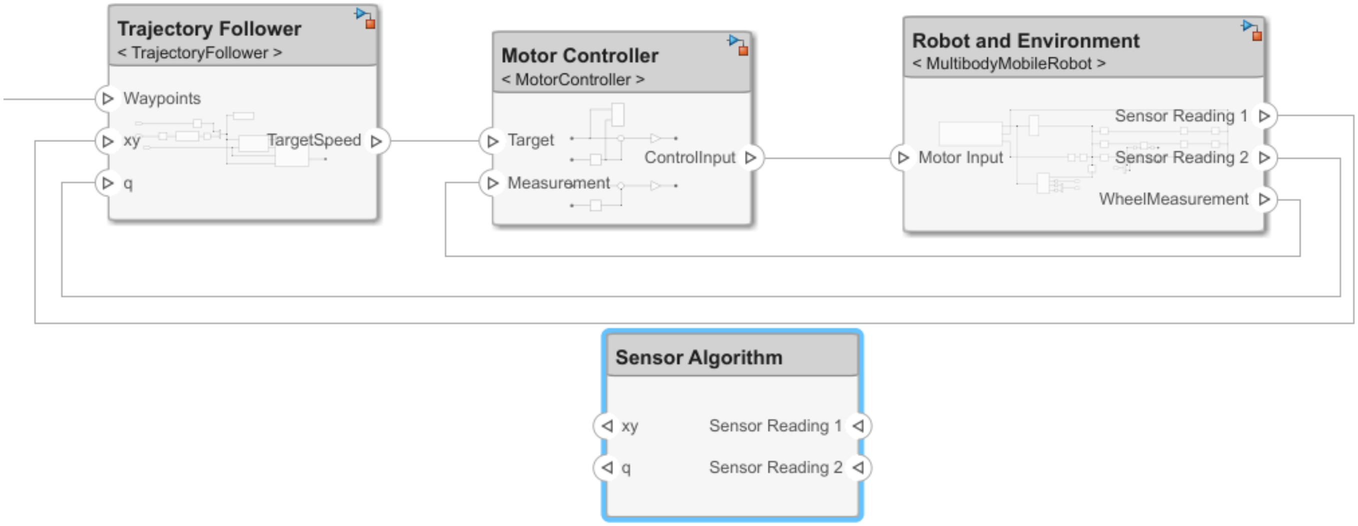
The structure of the logical architecture is similar to that of a Simulink model because simulation models are designed based on the flow of information. The components of the logical architecture model are linked to behavior models so that the architecture model can be simulated as well. Each component is responsible for one or more functions defined in the functional architecture model. `'Trajectory Follower'` is responsible for calculating the wheel speed of the robot based on the path the generator created. The lower level `'Motor Controller'` controls the speed of each actuator motor according to the output from the `'Trajectory Follower'`.

Note that many components are omitted from this example model. For example, sensor models like `Lidar Sensor` and `RGB Camera` are not required in this model because the true value from simulation gets the *x*-*y* position and orientation of the robot. For more complex simulations, sensor models like `RGB Camera` might be added to test different algorithms, such as object recognition. If such a sensor model was added, for example, `Lidar Sensor`, another behavior component would be required to decipher the sensor data, for example, `Scan Matching Algorithm`.

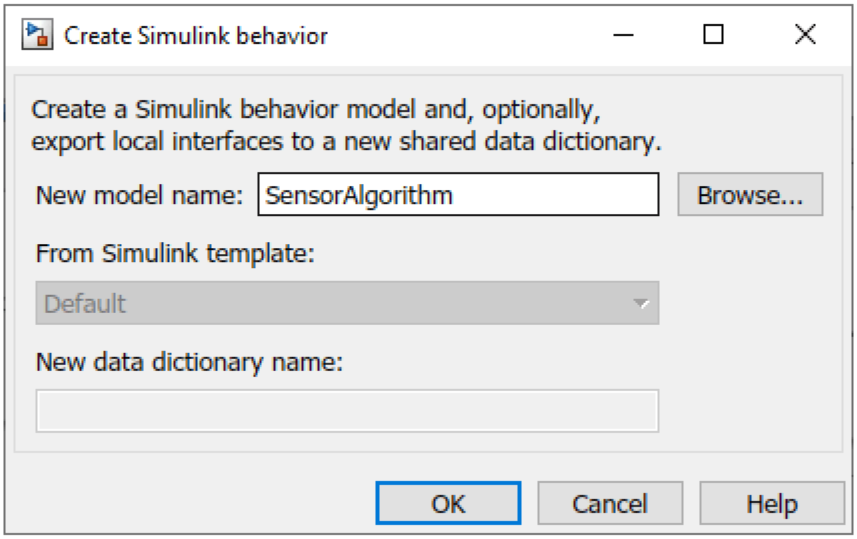
Add Simulink Behavior to Architecture Models with Bus Ports

Simulate an architecture model by adding Simulink behavior to a component.

Add a new component that would act as a sensor algorithm. Add two input ports and two output ports.



To create a new Simulink behavior, right-click **Create Simulink Behavior**.


Create Simulink behavior

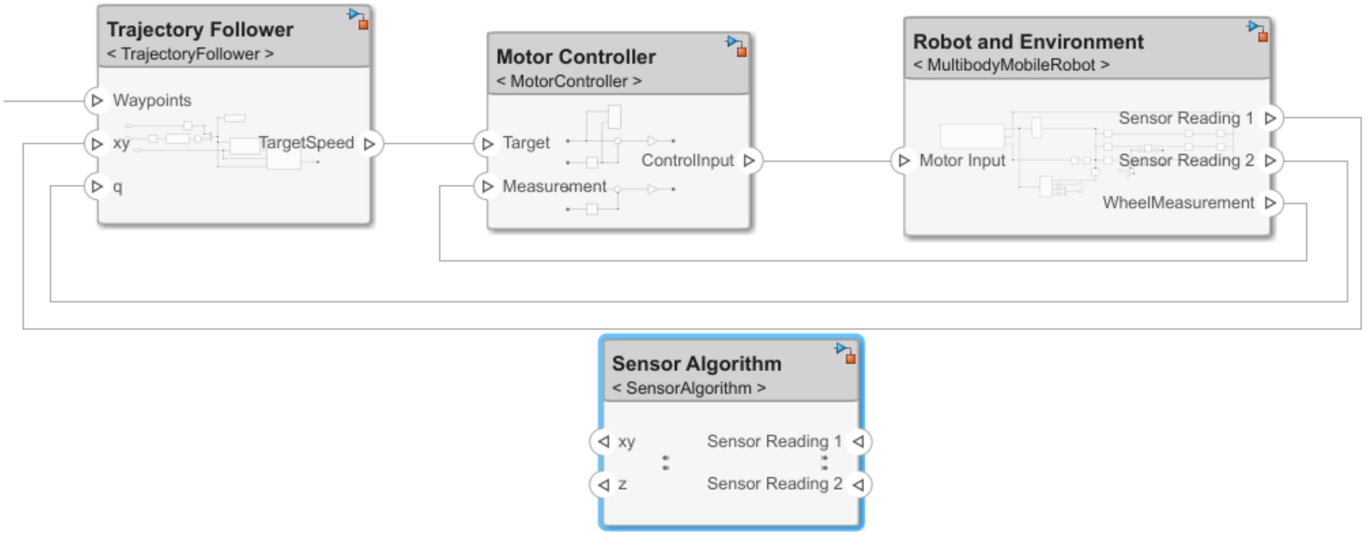
Create a Simulink behavior model and, optionally, export local interfaces to a new shared data dictionary.

New model name: SensorAlgorithm

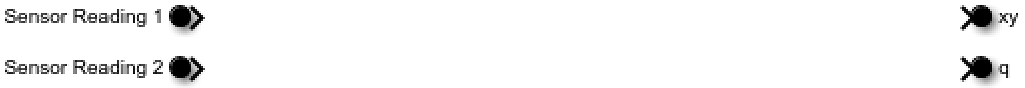
From Simulink template: Default

New data dictionary name:

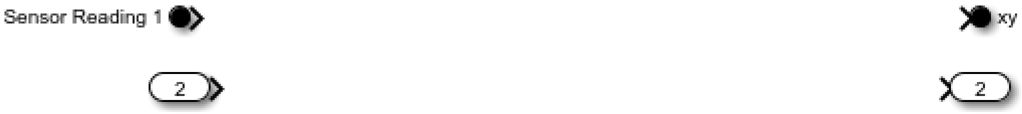
Click **OK**. The new Simulink model is saved under the current folder. The component `Component` is converted to a reference component called `Reference Component`.



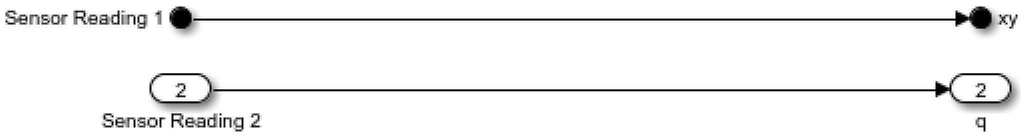
To edit the behavioral model, double-click `'Sensor Algorithm'`. Observe that bus element ports are created during the conversion process. For more information on setting bus ports, see "Explore Simulink Bus Capabilities".



Any port blocks can be used to connect different components, for example, `Inport` and `Outport`. Delete `'Sensor Reading 2'` and `'q'`. Create a new `Inport` and `Outport`.

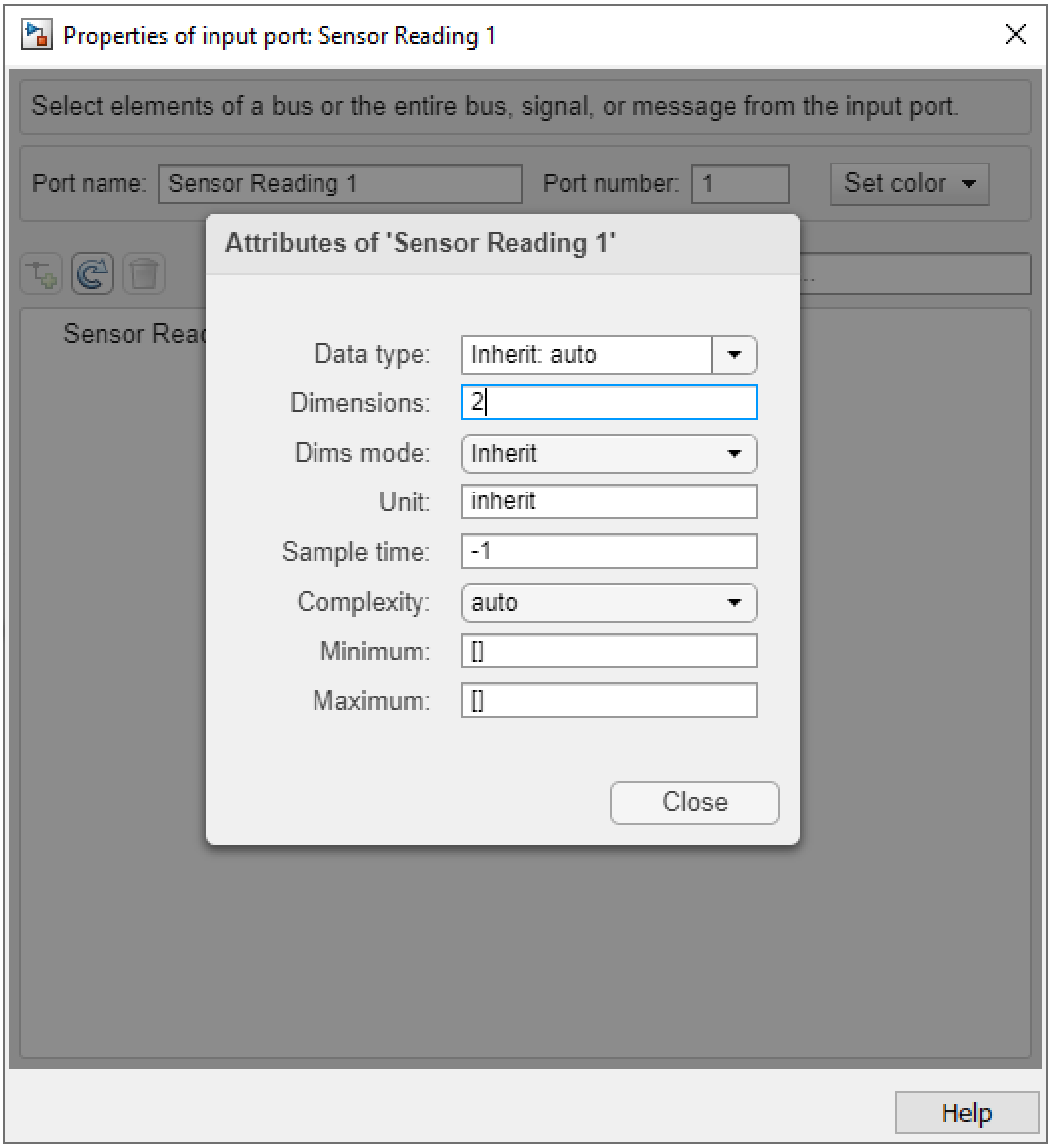


Connect the inports to the outports.



Note that in al scenario where sensor models such as `RGB Camera` and `Lidar Sensor` are added, the algorithm model will include tools like a neural network or scan matching method.

Click the pencil button to open **Block Parameters**. Set `'Sensor Reading 1'` `Dimensions` to 2.



Properties of input port: Sensor Reading 1

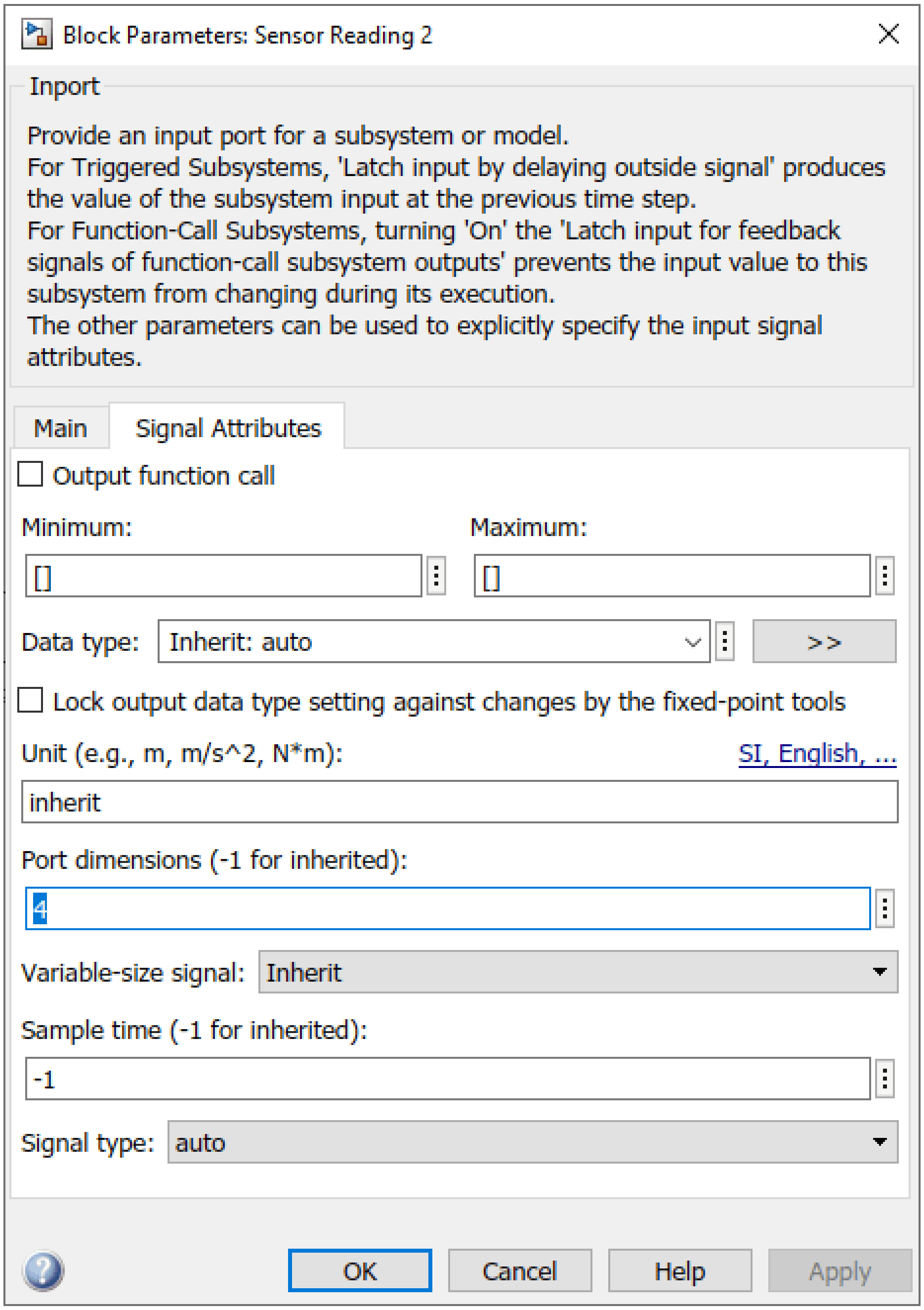
Select elements of a bus or the entire bus, signal, or message from the input port.

Port name: Sensor Reading 1
Port number: 1

Attributes of 'Sensor Reading 1'

Data type: Inherit: auto
Dimensions: 2
Dims mode: Inherit
Unit: inherit
Sample time: -1
Complexity: auto
Minimum: []
Maximum: []

Set `'Sensor Reading 2'` `Port dimensions` to `4`. Each port corresponds to an *x*-*y* position and quaternion, respectively.



Block Parameters: Sensor Reading 2

Inport

Provide an input port for a subsystem or model.
For Triggered Subsystems, 'Latch input by delaying outside signal' produces the value of the subsystem input at the previous time step.
For Function-Call Subsystems, turning 'On' the 'Latch input for feedback signals of function-call subsystem outputs' prevents the input value to this subsystem from changing during its execution.
The other parameters can be used to explicitly specify the input signal attributes.

□ Output function call

Minimum: []

Maximum: []

Data type: Inherit: auto

□ Lock output data type setting against changes by the fixed-point tools

Unit (e.g., m, m/s^2, N*m): inherit

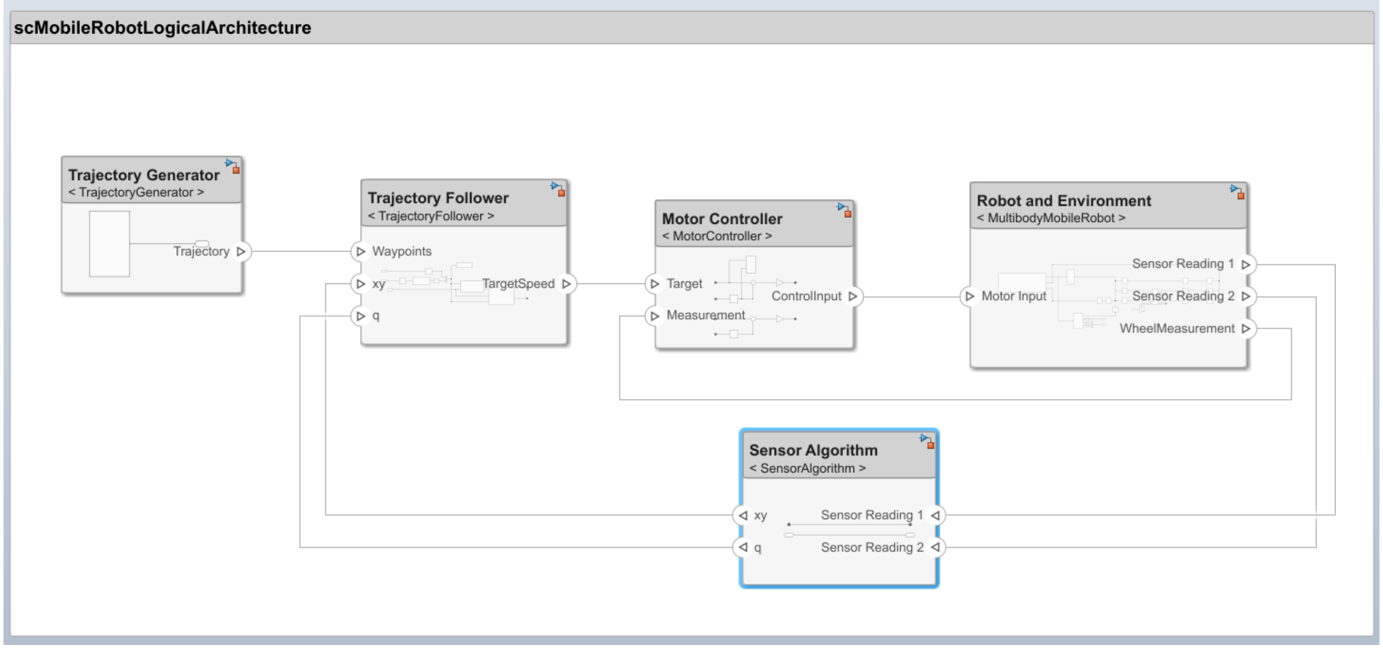
Port dimensions (-1 for inherited): 4

Variable-size signal: Inherit

Sample time (-1 for inherited): -1

Signal type: auto

Return to the logical architecture and connect the components.



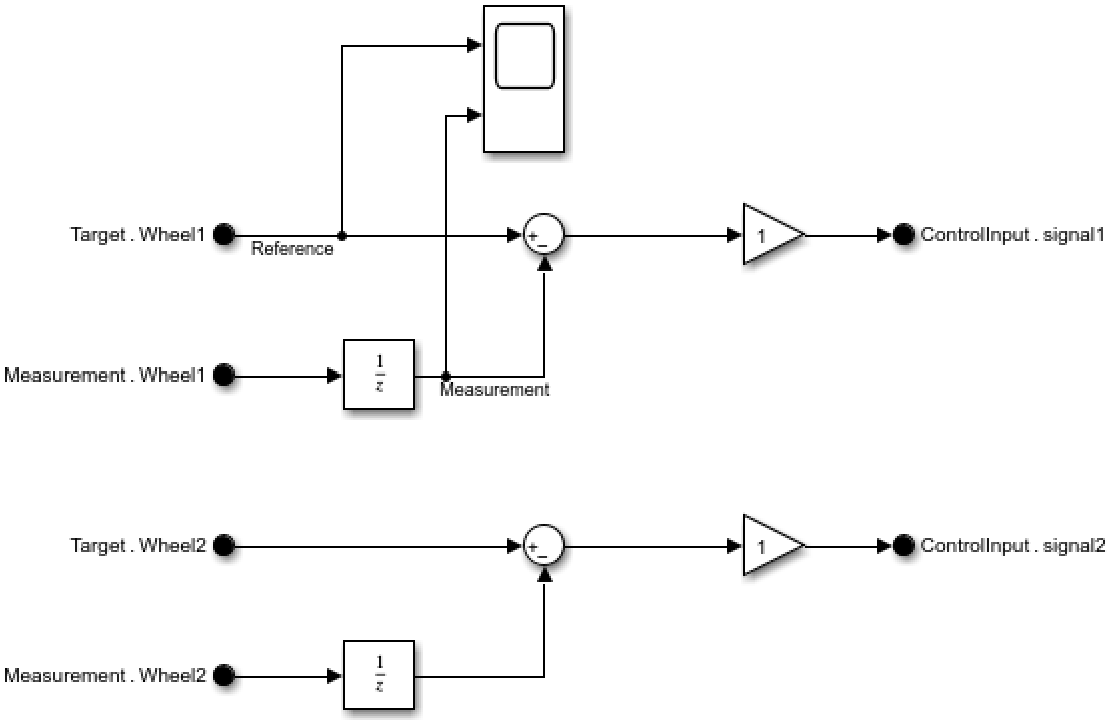
The process outlined above is already done in `scMobileRobotLogicalArchitecture`. Double-click the file to open the model, or enter the name in the MATLAB Command Window.

```
scMobileRobotLogicalArchitecture
```

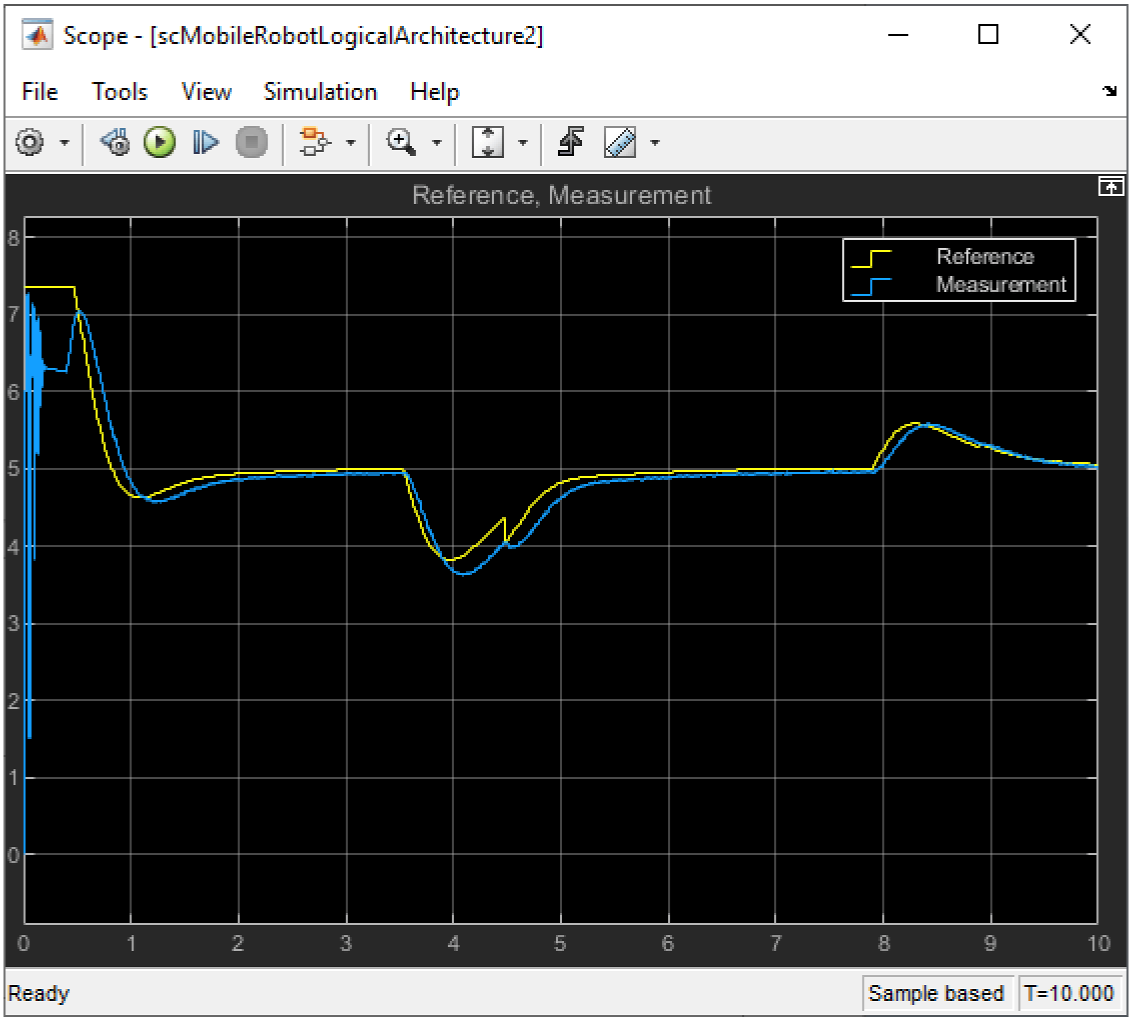
A behavior algorithm is created based on port information only. When designing a logical architecture, you can set the interface of the port in more detail. For example, if you know that a 800 x 600 RGB image with 24 frames per second will be transferred from the camera sensor, you can set the corresponding port interface accordingly to ensure efficient data transfer. For more information about setting interfaces, see “Define Interfaces” on page 3-2.

Running Simulation from Logical Architecture

Once behavior models are linked, the architecture model can be simulated just like any other Simulink models by clicking **Run**.

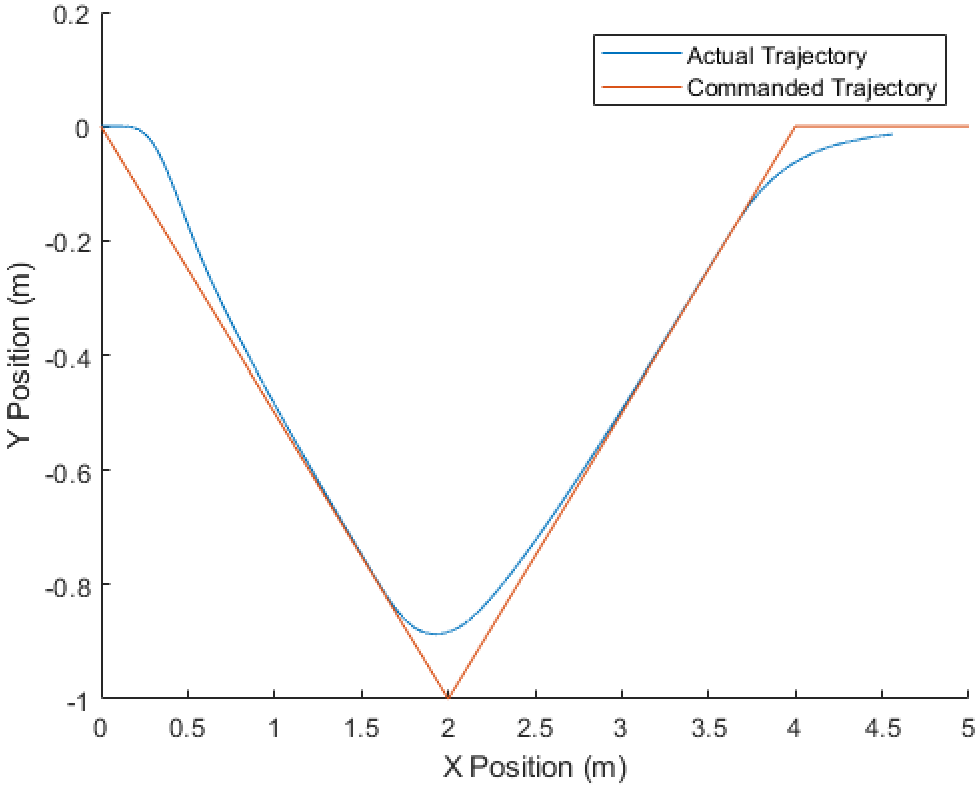


The scope from `'MotorController'` shows how well a simple P-gain controller is performing to follow the reference velocity for one of the wheels on the robot.



Run the following script to observe you well the robot follows the waypoints.

```
out = sim('scMobileRobotLogicalArchitecture.slx');
% waypoints are manually defined in Constant block
waypoints = eval(get_param('TrajectoryGenerator/Manual Waypoints','Value'));

figure
hold on
plot(out.pose.Data(:,1),out.pose.Data(:,2))
plot(waypoints(:,1),waypoints(:,2))
hold off
xlabel('X Position (m)')
ylabel('Y Position (m)')
legend('Actual Trajectory','Commanded Trajectory')
```



See Also

More About

- “Model-Based Design with Simulink”
- “Requirement Links” (Simulink Requirements)
- “Create an Architecture Model” on page 1-2
- “Link and Trace Requirements” on page 6-25
- “Allocate Architectures in a Tire Pressure Monitoring System” on page 6-5
- “Define Profiles and Stereotypes” on page 4-2
- “Use Stereotypes and Profiles” on page 4-10
- “Analyze Architecture” on page 6-10
- “Explore Simulink Bus Capabilities”
- “Define Interfaces” on page 3-2

7

Software Architectures

Author Software Architectures

Software architectures in System Composer provide capabilities to author software architecture models composed of software components, ports, and interfaces. Use System Composer to design your software architecture model, simulate your design in the architecture level, and generate code.

Use software architectures to link your Simulink Export-Function, Rate-based or JMAAB models to components in your architecture model to simulate and generate code.

Create a Software Architecture Model

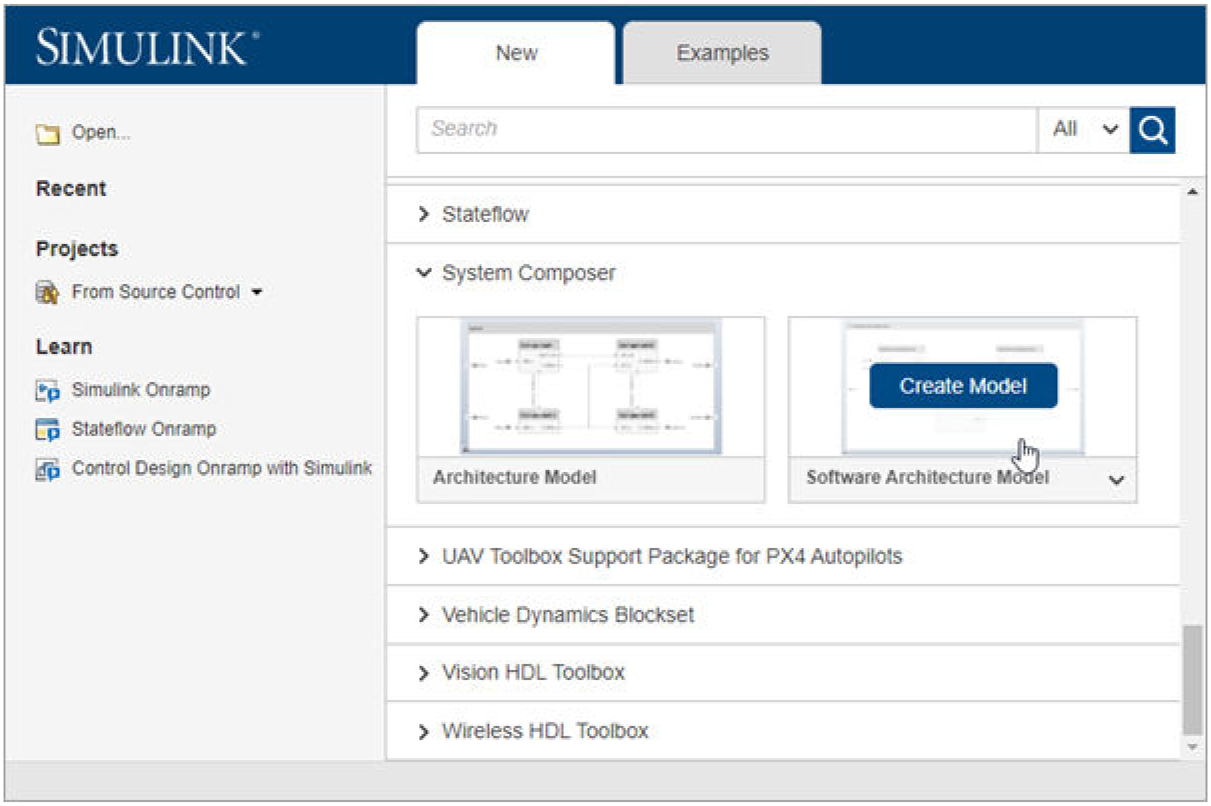
The workflow for authoring software architecture models is similar to authoring system architectures. Start with a blank software architecture template to model.

You can create a software architecture programmatically by using the function.

```
systemcomposer.createModel('mySoftwareArchitectureDesign', 'SoftwareArchitecture'),
```

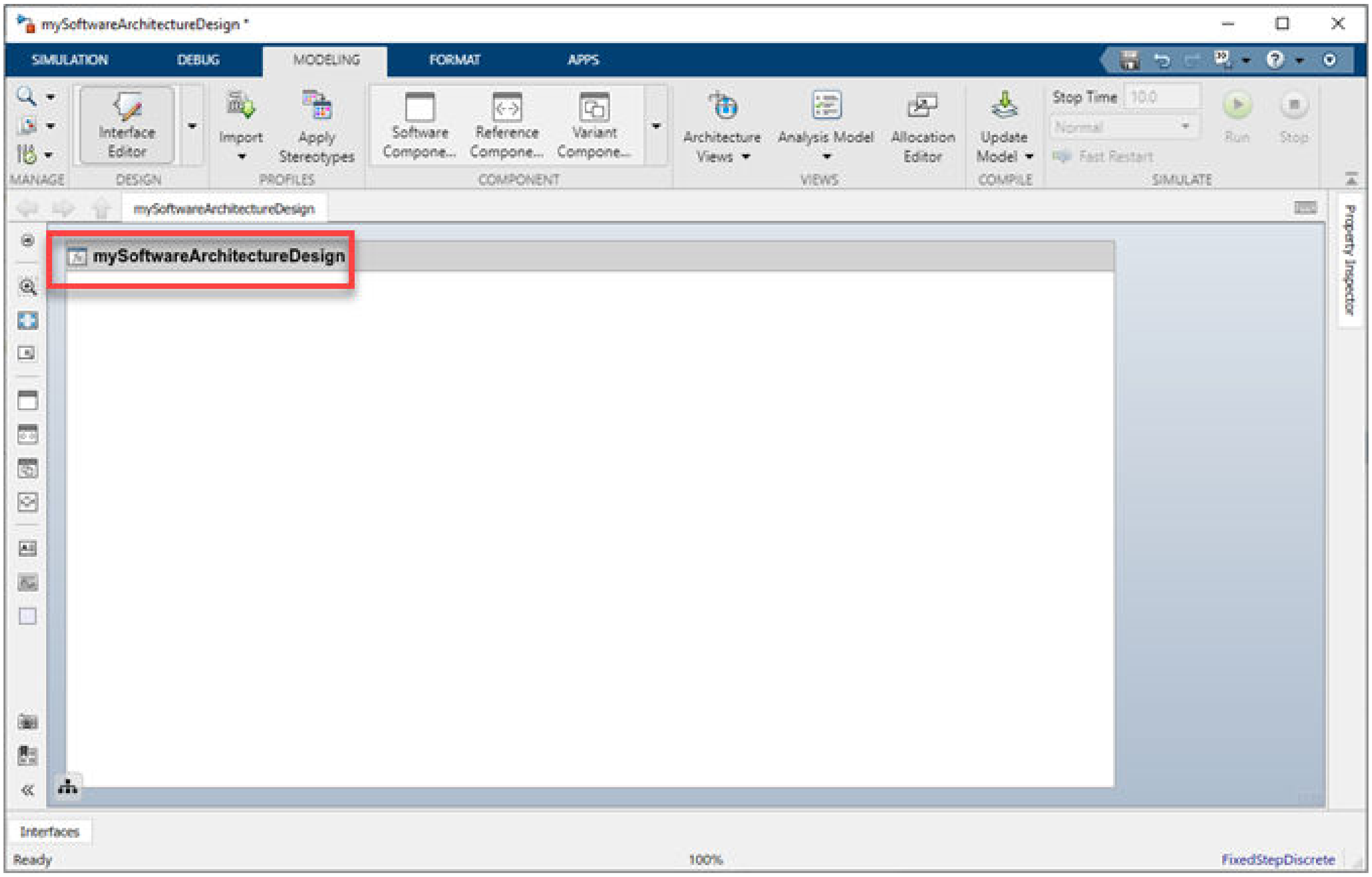
where `mySoftwareArchitectureDesign` is the name of the new model.

You can also use the provided template in the Simulink start page.



From a Simulink model or a System Composer architecture model, on the **Simulation** tab, select **New**, and then select **Architecture**. Then, select **Software Architecture Model**.

System Composer opens a new empty software architecture model. Observe the icon on the upper left corner that distinguishes the empty model from a system architecture.

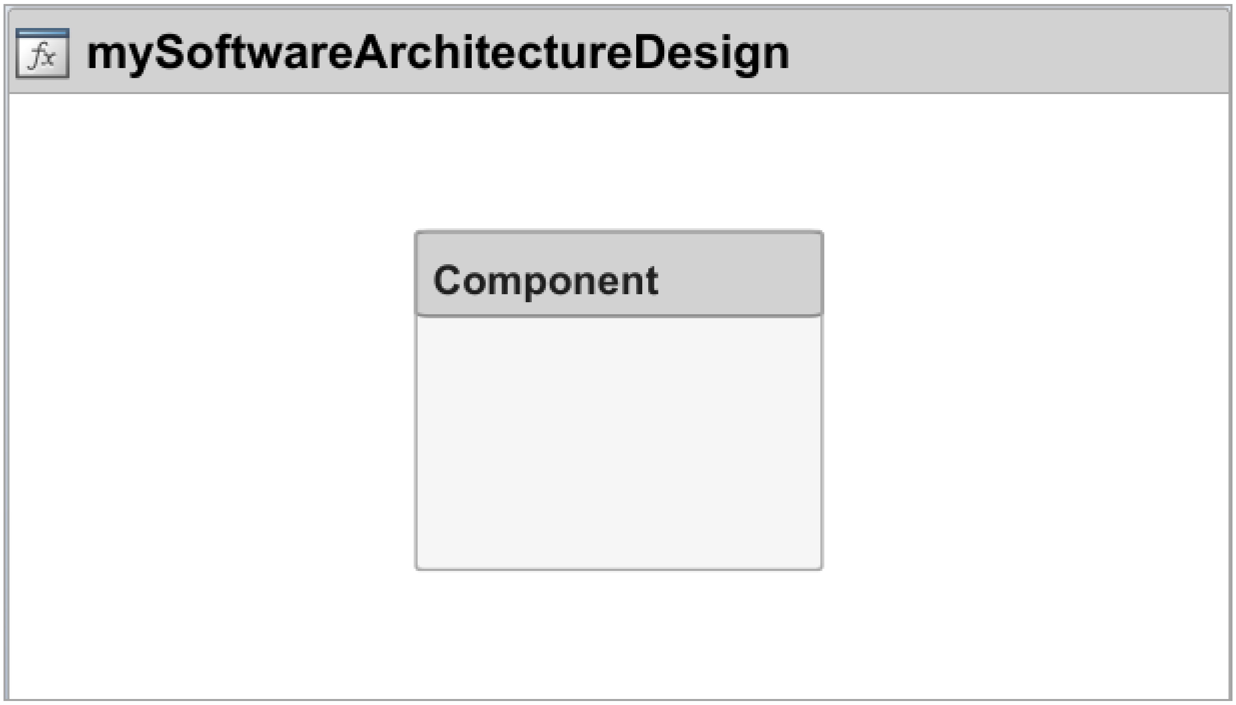


When you model software architectures, you can:

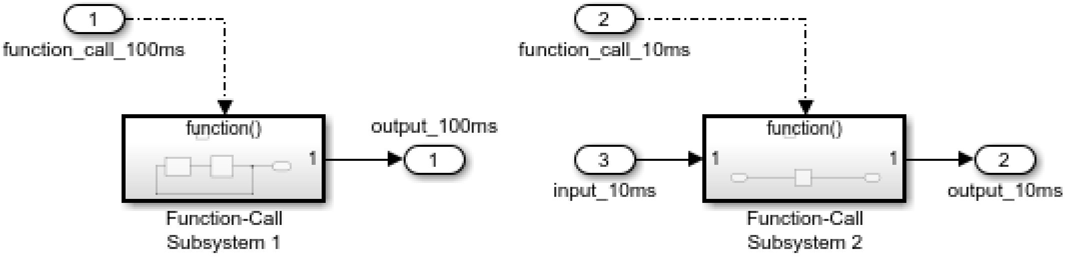
- Use model building and visualization tools provided by System Composer such as components, connections, and ports. For more information, see “Compose Architecture Visually” on page 1-2.
- Define interfaces. For more information, see “Define Interfaces” on page 3-2.
- Create custom views. For more information, see “Create Architecture Views Interactively” on page 1-37.
- Use tools to write analysis and create allocations. For more information, see “Analyze Architecture” on page 6-10 and “Create and Manage Allocations” on page 6-2.

Build a Simple Software Architecture Model

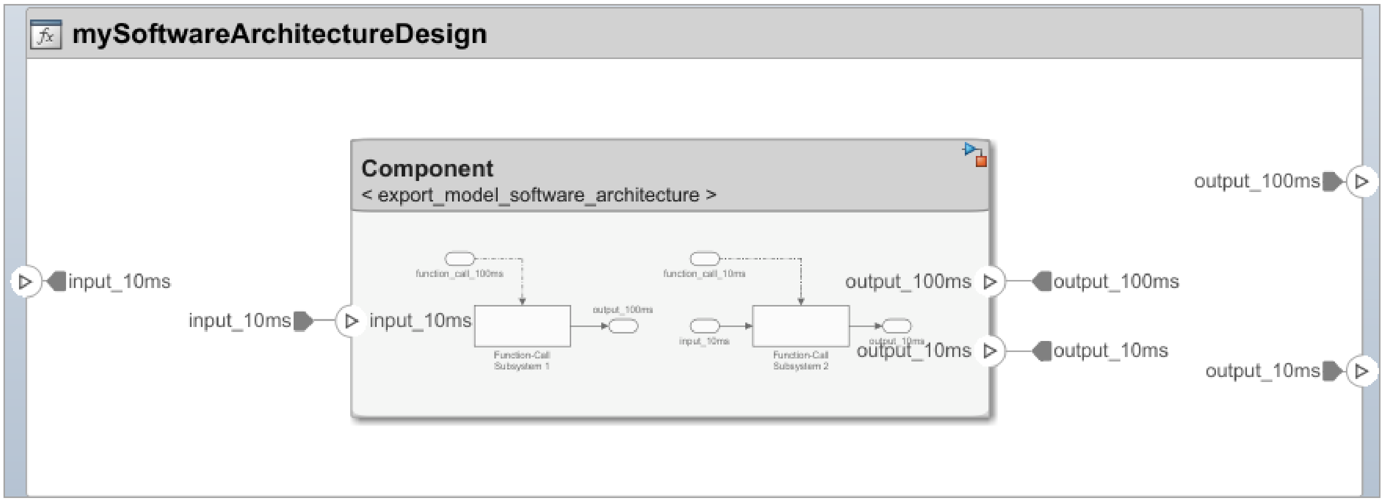
1 Drag an empty component to the `mySoftwareArchitectureDesign` model.



2 Link this simple Simulink Export-Function model, `export_model_software_architecture` to your component by right-clicking the component and selecting **Link to model**. For more information about building this Simulink model, see “Create an Export-Function Model”.



3 Connect component input port and output ports to architecture input ports and output ports.



In this example, you start from a blank template and create a simple software architecture model. To learn how to simulate a software architecture model and generate code, see “Simulate and Deploy Software Architectures” on page 7-6.

See Also

More About

- “Compose Architecture Visually” on page 1-2
- “Create an Export-Function Model”

Simulate and Deploy Software Architectures

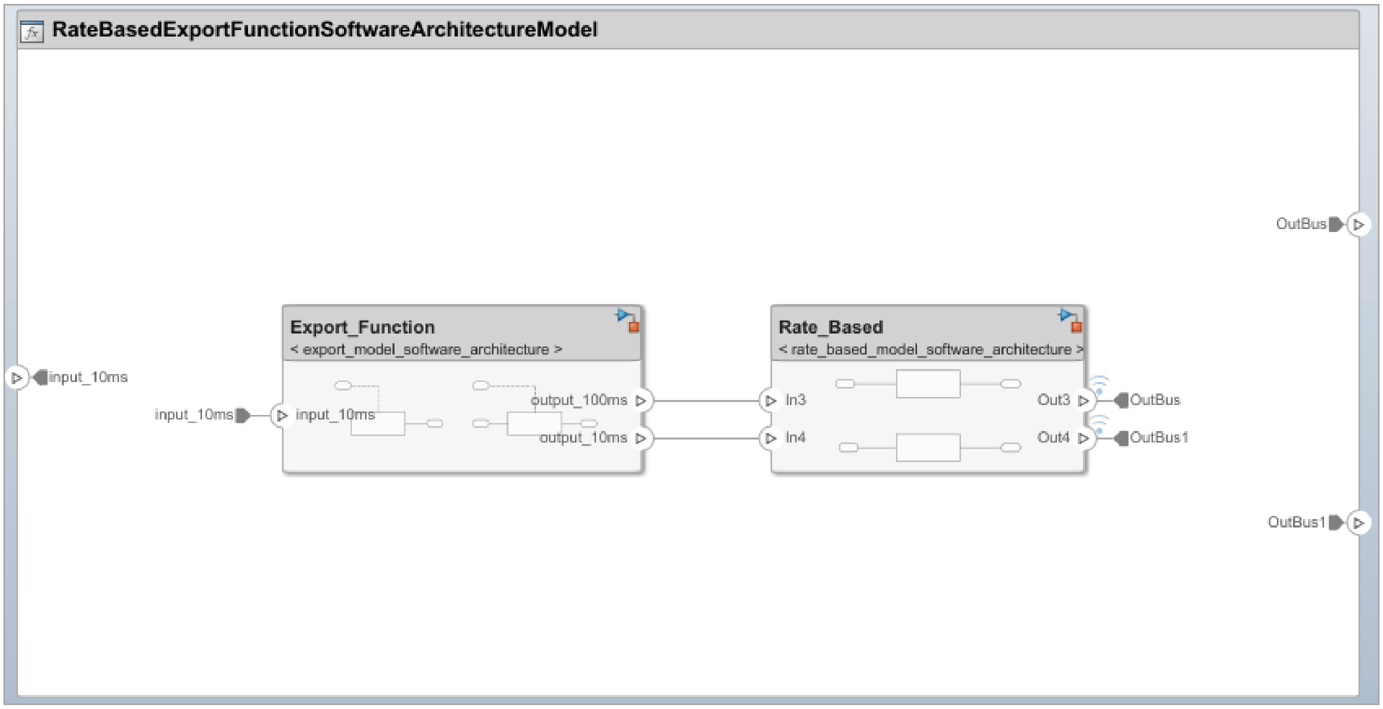
This example shows how to build a multi-component software architecture model with a rate-based and export-function components, how to simulate your design at the architecture level, and how to generate code.

Open the Software Architecture Model

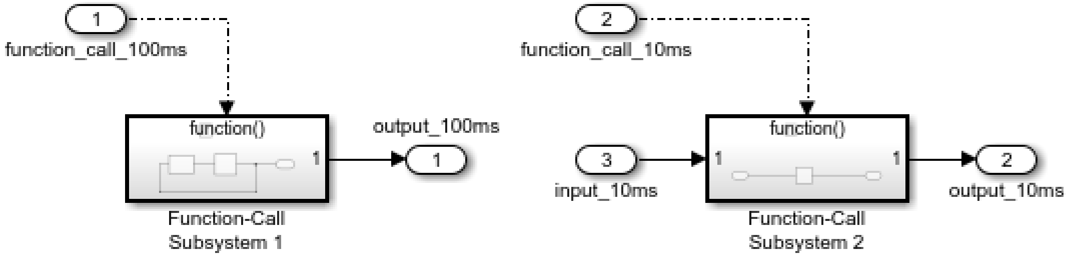
This software architecture model has two software components: Export_Function and Rate_Based.

```
open_system('RateBasedExportFunctionSoftwareArchitectureModel')
```

In the software architecture model, the Export_Function component is linked to a Simulink® export-function behavior model, `export_model_software_architecture`.

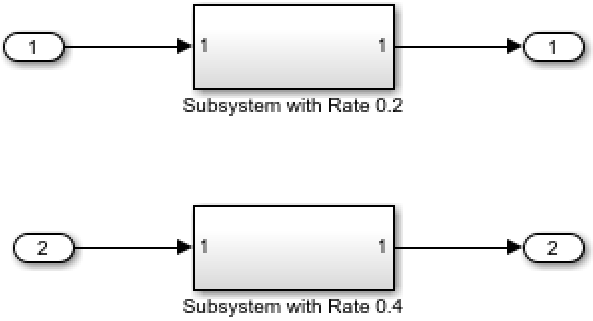

RateBasedExportFunctionSoftwareArchitectureModel
OutBus
input_10ms
Export_Function
< export_model_software_architecture >
input_10ms
input_10ms
output_100ms
output_10ms
Rate_Based
< rate_based_model_software_architecture >
In3
In4
Out3
Out4
OutBus
OutBus1
OutBus1

In this Simulink behavior, two functions are modeled using Function-Call Subsystem blocks. The inport blocks are connected to the function-call input ports and generate periodic function-call events with sample times `10ms` and `100ms`. To learn how to model this behavior, see “Create an Export-Function Model”.



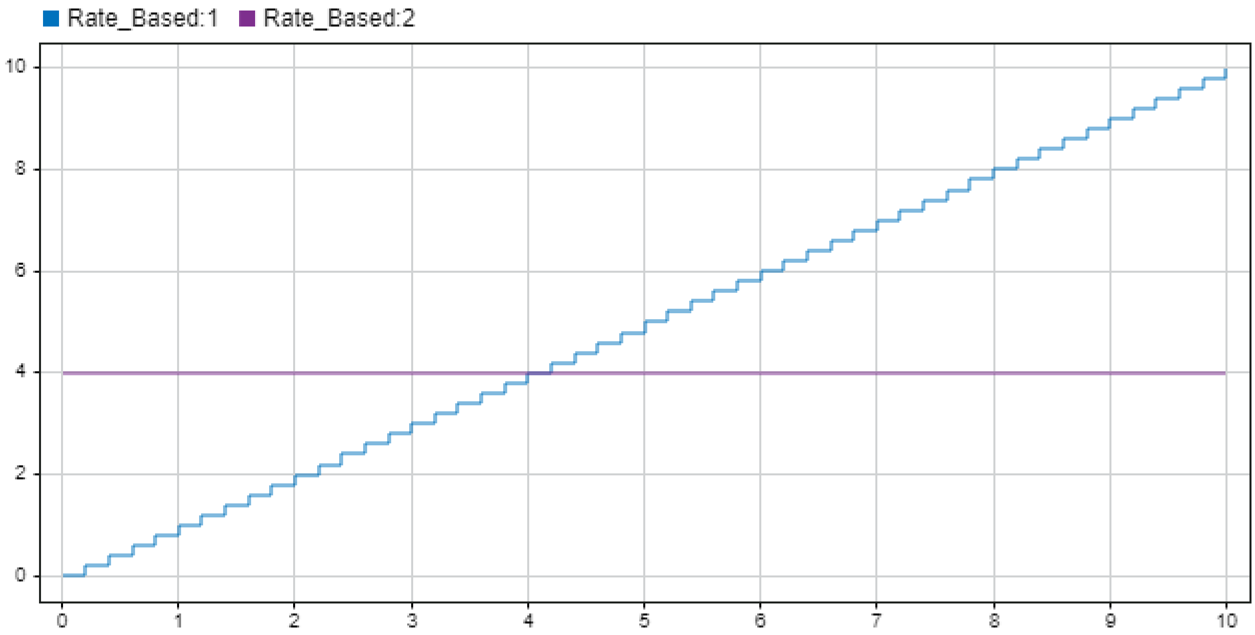
If the inport blocks that are connected to the function-call input ports with sample time specified as -1, meaning the functions are aperiodic, use a Simulink test model with explicit scheduling blocks such as a Stateflow chart to simulate. For more information see Test Software Architecture on page 7-0 .

The Rate_Based component is linked to `rate_based_model_software_architecture` as the Simulink behavior model. To learn how to create this rate-based model, see “Create A Rate-Based Model”.

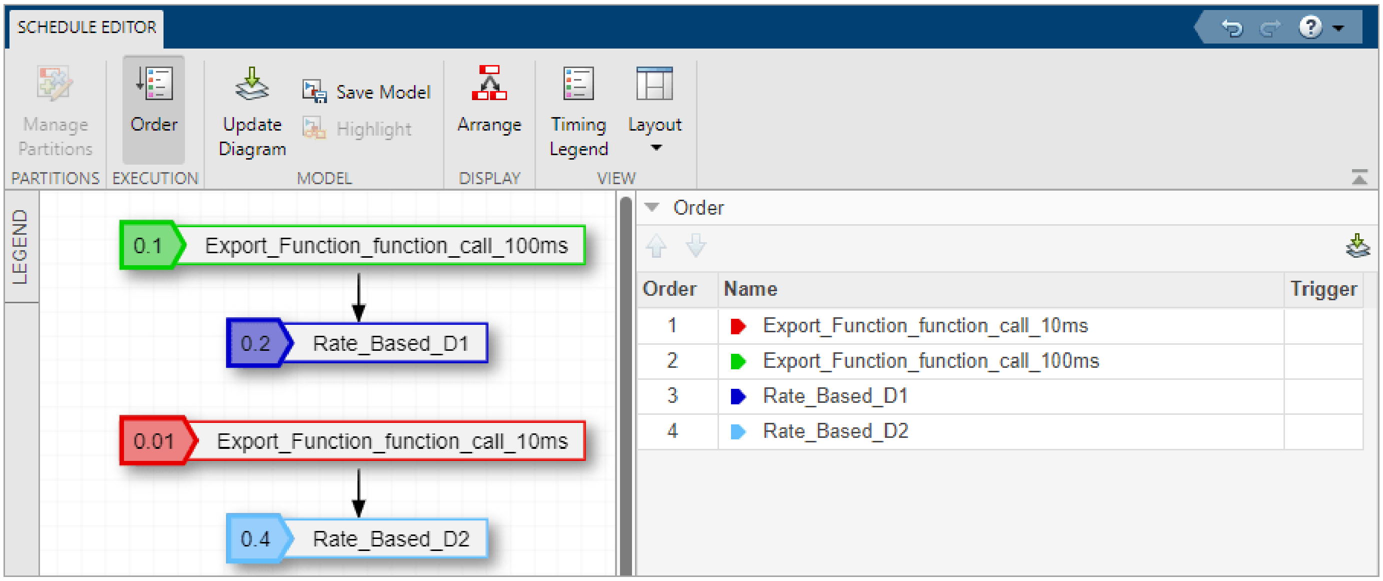


Simulate the Model with Default Execution Order

Simulate the model. Observe that the Simulation Data Inspector displays the output from the Rate-Based component.



To see and change the default execution order of the functions, you can use the Scheduling Editor. For more information, see “Using the Schedule Editor”.



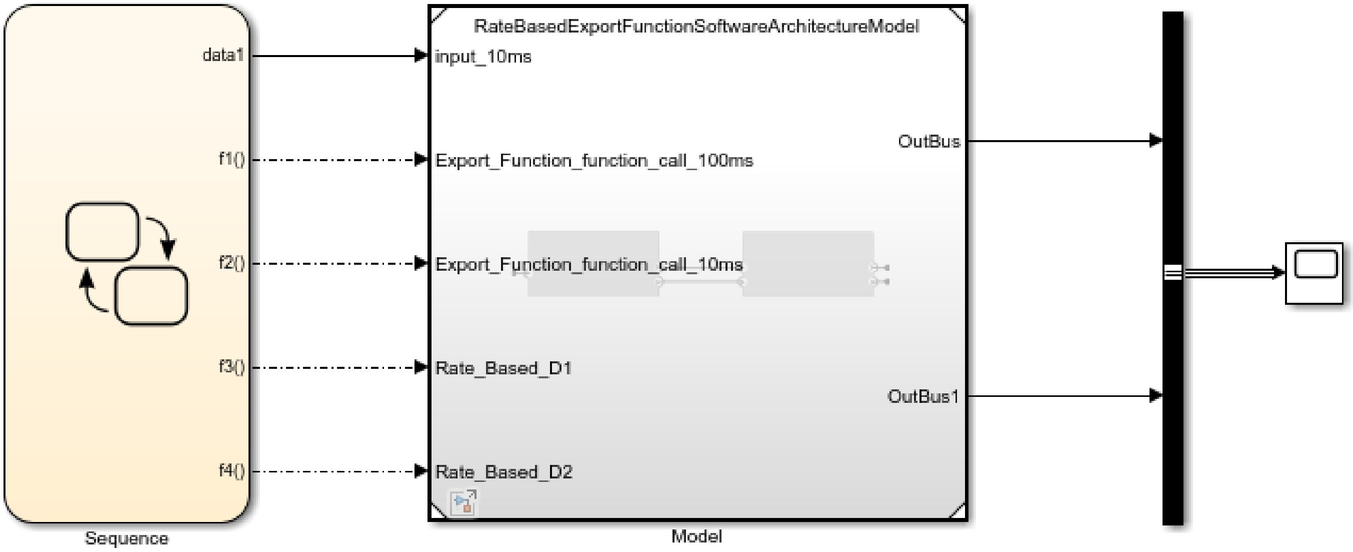
Order	Name	Trigger
1	🟥 Export_Function_function_call_10ms	
2	🟩 Export_Function_function_call_100ms	
3	🟦 Rate_Based_D1	
4	🟦 Rate_Based_D2	

Test Software Architecture

You can test a software architecture model and simulate different execution orders of functions by referencing it from a Model block in a Simulink test model with explicit scheduling blocks such as Stateflow® Chart (Stateflow).

In this example, a Model block that references a software architecture model has a function-call input port for each function in the architecture model.

To simulate the architecture model with a Stateflow chart periodic scheduler, connect the Stateflow chart function-call outputs to the Model block function-call inputs.

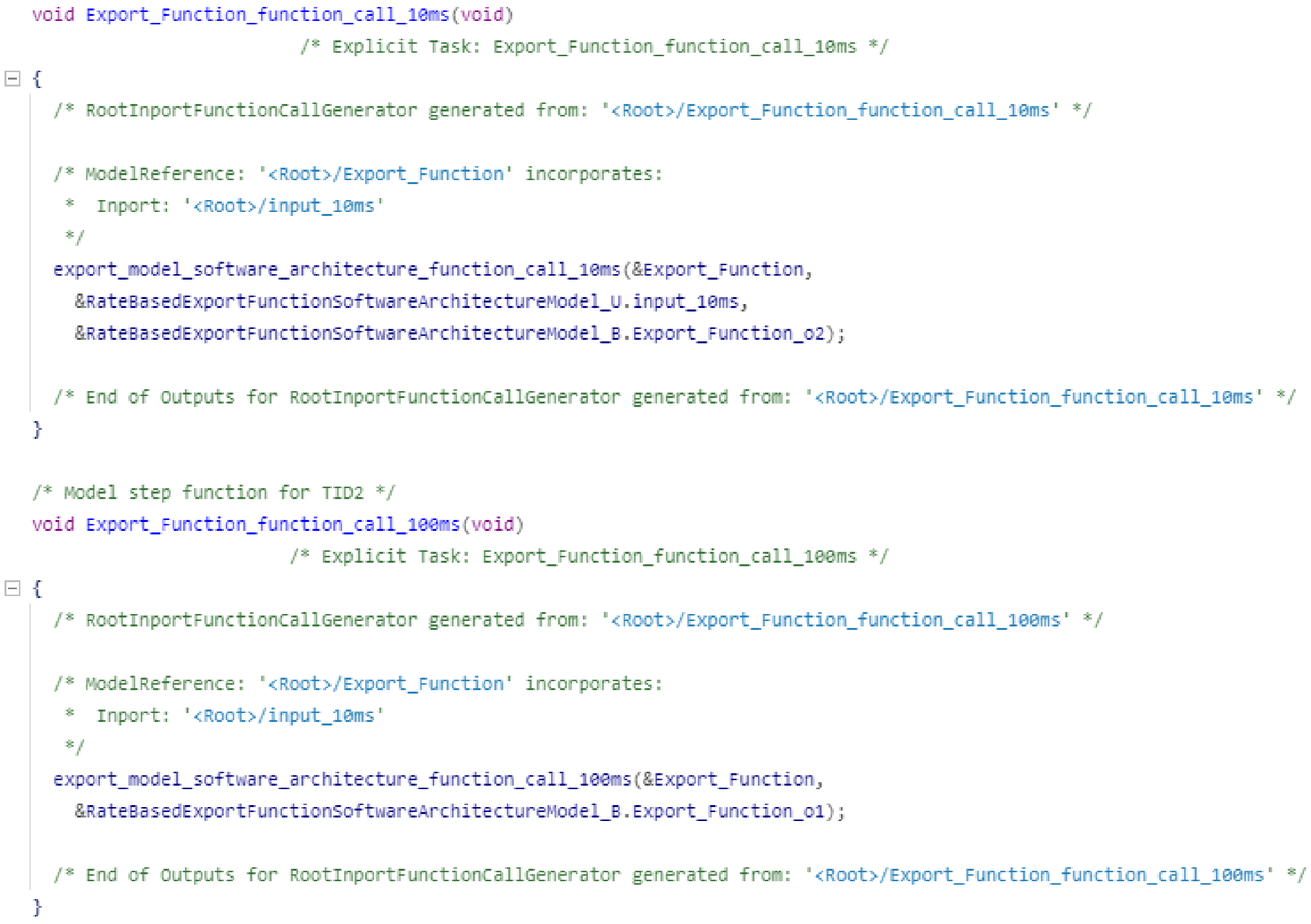

data1
f1()
f2()
f3()
f4()
Sequence
RateBasedExportFunctionSoftwareArchitectureModel
input_10ms
Export_Function_function_call_100ms
Export_Function_function_call_10ms
Rate_Based_D1
Rate_Based_D2
OutBus
OutBus1
Model

Deploy Software Architecture

You can generate code from the software architecture model for the functions of the export-function and rate-based components.

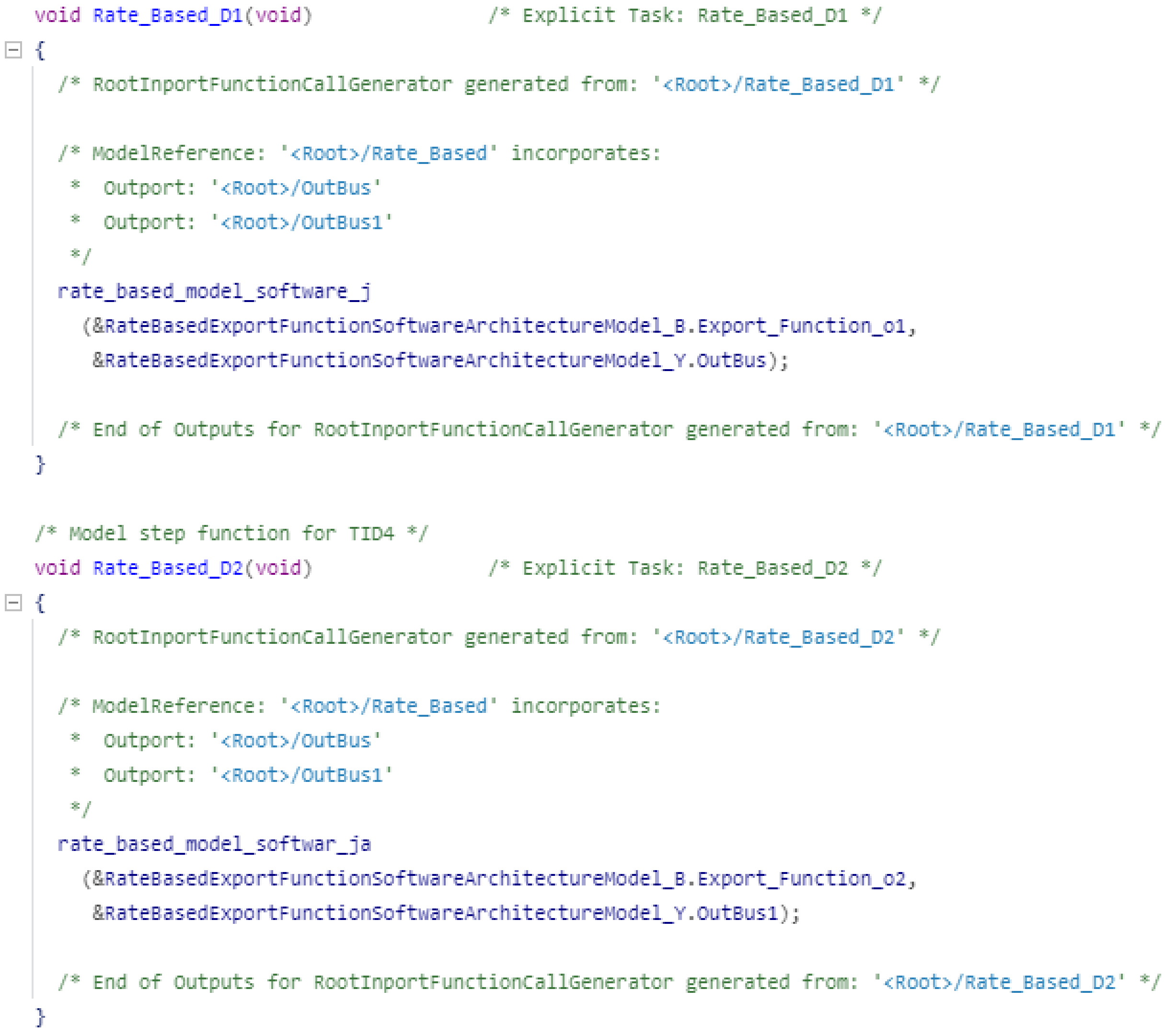
To generate code, from the **Apps** tab, select **Embedded Coder**. On the **C Code** tab, select **Generate Code**. The generated code contains an entry-point for each function of the component. For more information, see “Generate Code for Export-Function Model”.

For the export-function component, it generated the two functions that correspond to the function-call inport blocks inside the referenced export-function model.



```
void Export_Function_function_call_10ms(void)
                         /* Explicit Task: Export_Function_function_call_10ms */
{
  /* RootInportFunctionCallGenerator generated from: '<Root>/Export_Function_function_call_10ms' */

  /* ModelReference: '<Root>/Export_Function' incorporates:
   *  Inport: '<Root>/input_10ms'
   */
  export_model_software_architecture_function_call_10ms(&Export_Function,
    &RateBasedExportFunctionSoftwareArchitectureModel_U.input_10ms,
    &RateBasedExportFunctionSoftwareArchitectureModel_B.Export_Function_o2);

  /* End of Outputs for RootInportFunctionCallGenerator generated from: '<Root>/Export_Function_function_call_10ms' */
}

/* Model step function for TID2 */
void Export_Function_function_call_100ms(void)
                        /* Explicit Task: Export_Function_function_call_100ms */
{
  /* RootInportFunctionCallGenerator generated from: '<Root>/Export_Function_function_call_100ms' */

  /* ModelReference: '<Root>/Export_Function' incorporates:
   *  Inport: '<Root>/input_10ms'
   */
  export_model_software_architecture_function_call_100ms(&Export_Function,
    &RateBasedExportFunctionSoftwareArchitectureModel_B.Export_Function_o1);

  /* End of Outputs for RootInportFunctionCallGenerator generated from: '<Root>/Export_Function_function_call_100ms' */
}
```

Observe that, each rate-based component has separate entry point functions that correspond to each sample time in the referenced rate based model.



```
void Rate_Based_D1(void)               /* Explicit Task: Rate_Based_D1 */
{
  /* RootInportFunctionCallGenerator generated from: '<Root>/Rate_Based_D1' */

  /* ModelReference: '<Root>/Rate_Based' incorporates:
   *  Outport: '<Root>/OutBus'
   *  Outport: '<Root>/OutBus1'
   */
  rate_based_model_software_j
    (&RateBasedExportFunctionSoftwareArchitectureModel_B.Export_Function_o1,
     &RateBasedExportFunctionSoftwareArchitectureModel_Y.OutBus);

  /* End of Outputs for RootInportFunctionCallGenerator generated from: '<Root>/Rate_Based_D1' */
}

/* Model step function for TID4 */
void Rate_Based_D2(void)               /* Explicit Task: Rate_Based_D2 */
{
  /* RootInportFunctionCallGenerator generated from: '<Root>/Rate_Based_D2' */

  /* ModelReference: '<Root>/Rate_Based' incorporates:
   *  Outport: '<Root>/OutBus'
   *  Outport: '<Root>/OutBus1'
   */
  rate_based_model_softwar_ja
    (&RateBasedExportFunctionSoftwareArchitectureModel_B.Export_Function_o2,
     &RateBasedExportFunctionSoftwareArchitectureModel_Y.OutBus1);

  /* End of Outputs for RootInportFunctionCallGenerator generated from: '<Root>/Rate_Based_D2' */
}
```

See Also

More About

- “Author Software Architectures” on page 7-2
- “Compose Architecture Visually” on page 1-2
- “Create an Export-Function Model”
- “Create A Rate-Based Model”

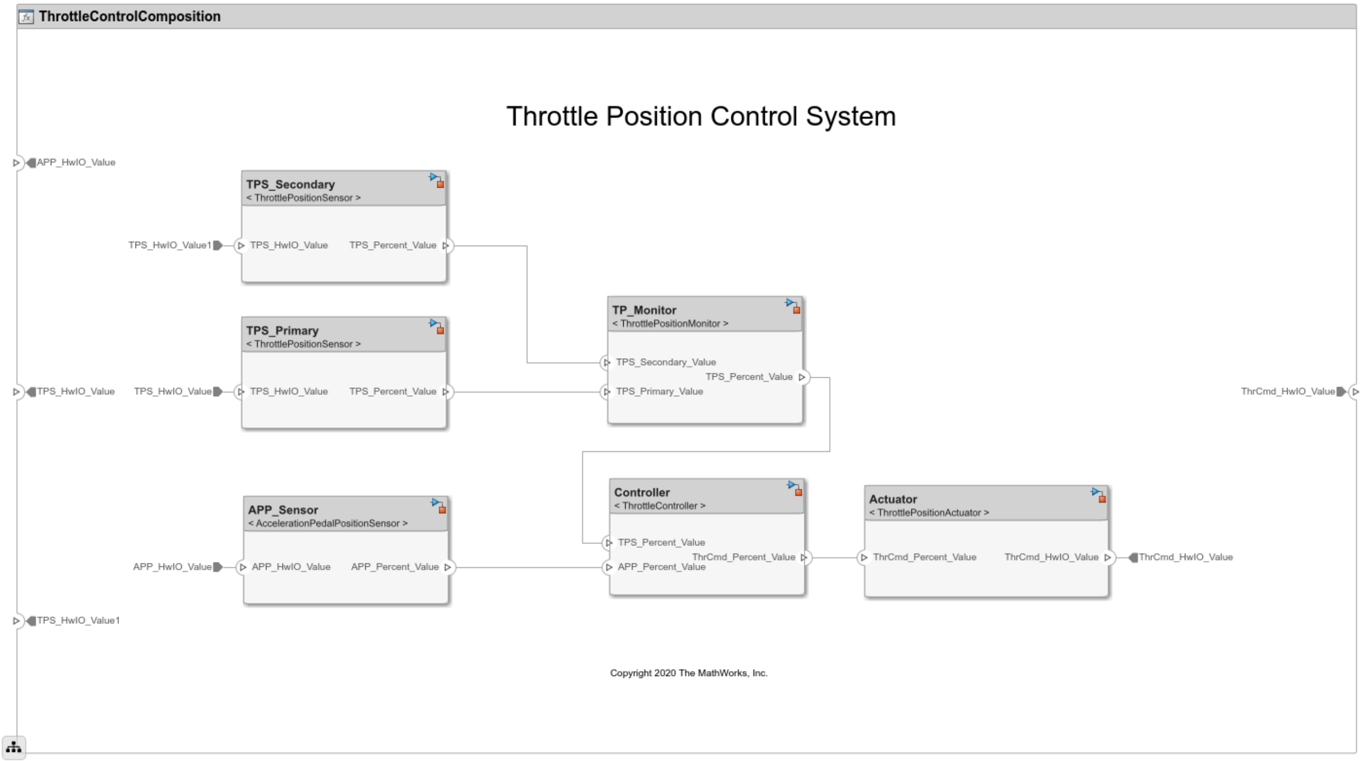
Modeling the Software Architecture of a Throttle Position Control System

This example shows how to author the software architecture of a throttle position control system in System Composer®, schedule and simulate the execution order of the functions from its components, and generate code.

Throttle Control Composition

In this example, the software architecture of a throttle position control system is modeled in System Composer using six components. The throttle position control component reads the throttle and pedal positions and outputs the new throttle position. Two throttle position sensor components provide the current position of the throttle, and a pedal position sensor component provides the applied pedal position. These signals are used by a controller component to determine the new throttle position.

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

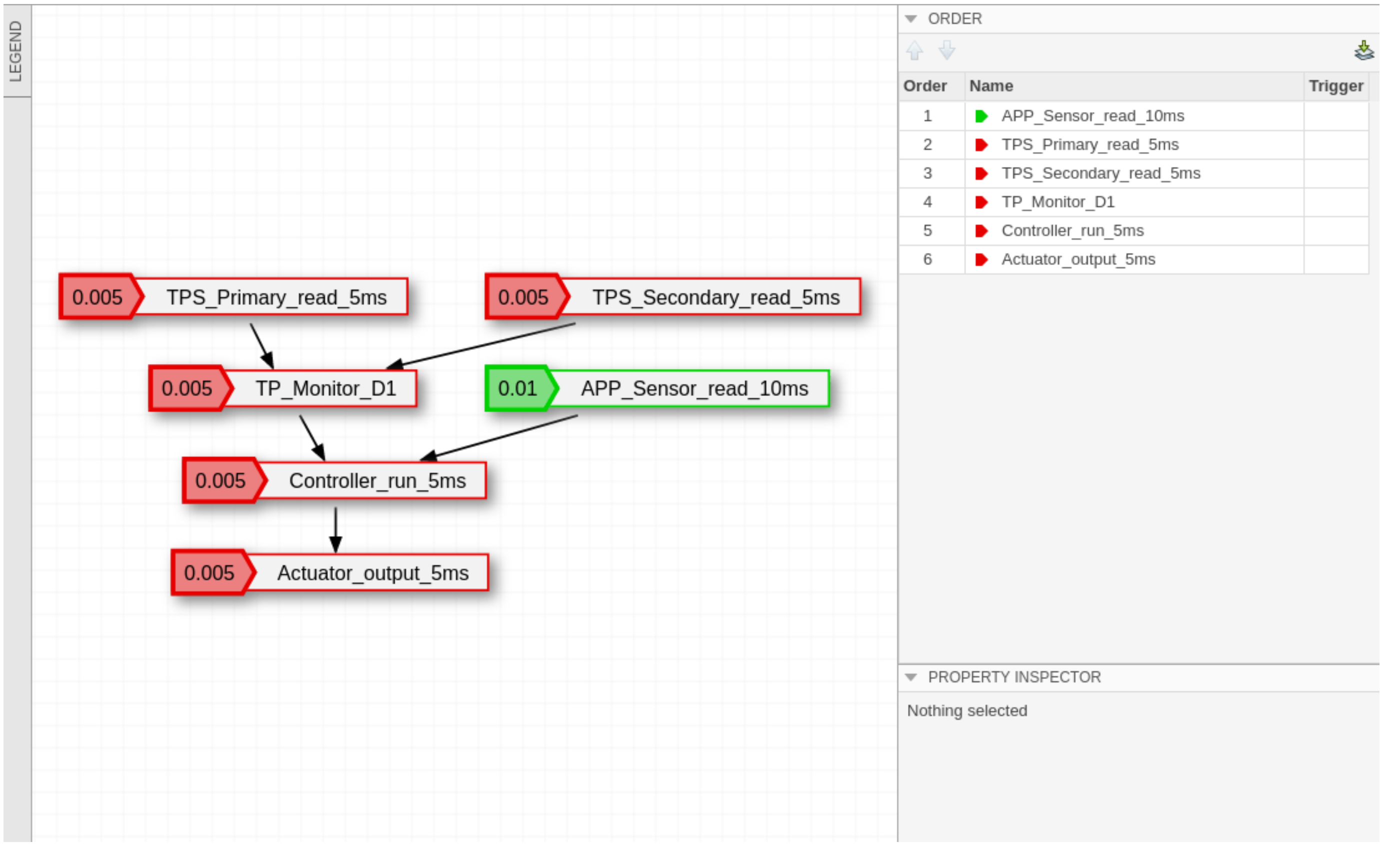


Simulate the Model at the Architecture Level

Simulate the software architecture model.

```
sim('ThrottleControlComposition');
```

To view and change the default execution order of the functions from the components, use the Schedule Editor. To open the Schedule Editor, on the **Modeling** tab, in the **Design** section, click **Schedule Editor**. For more information about scheduling functions with the Schedule Editor, see Schedule an Export-Function Model Using the Schedule Editor.



ORDER

Order	Name	Trigger
1	🟩 APP_Sensor_read_10ms	
2	🟥 TPS_Primary_read_5ms	
3	🟥 TPS_Secondary_read_5ms	
4	🟥 TP_Monitor_D1	
5	🟥 Controller_run_5ms	
6	🟥 Actuator_output_5ms	

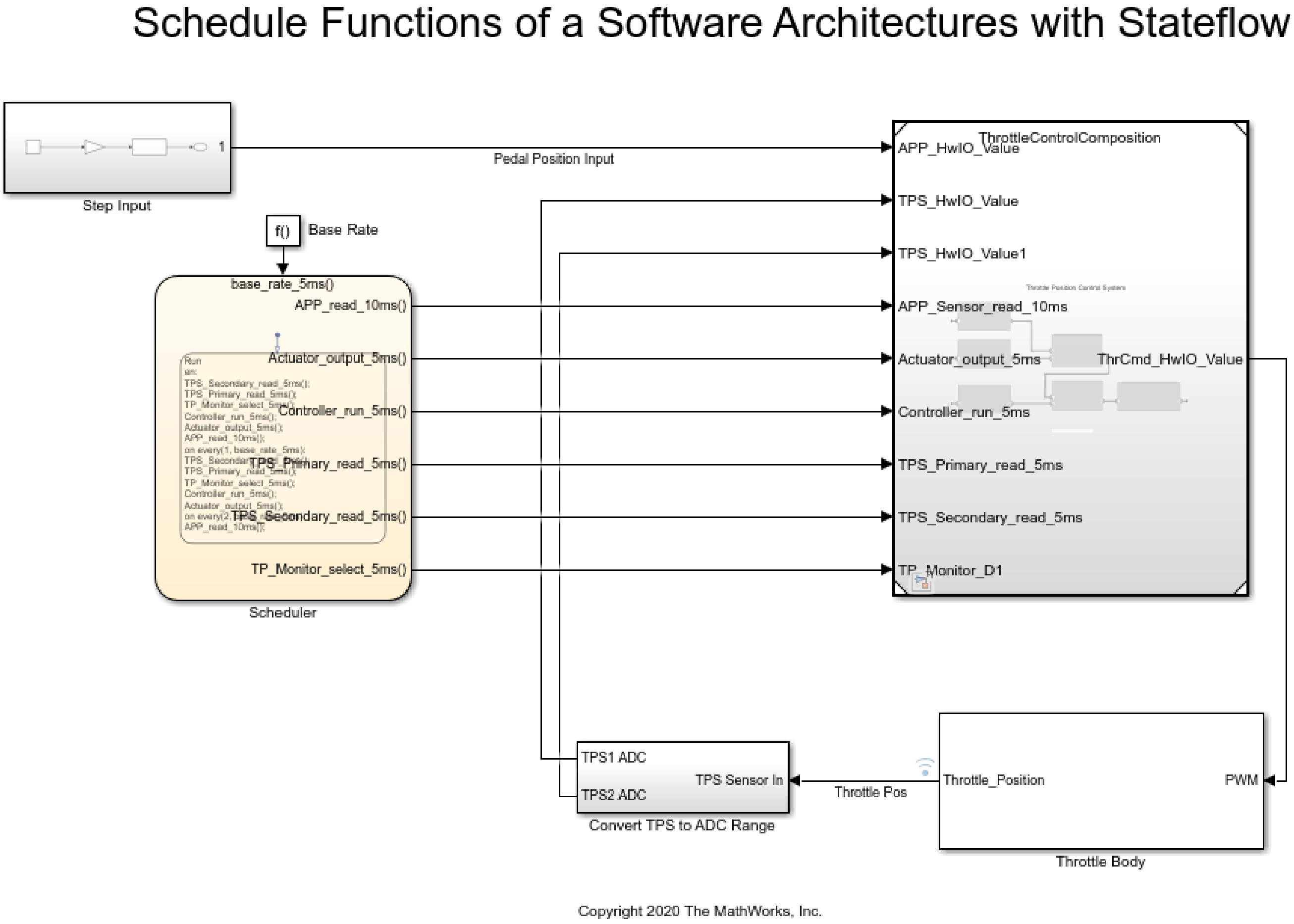
PROPERTY INSPECTOR

Nothing selected

Simulate the Model at the System Level

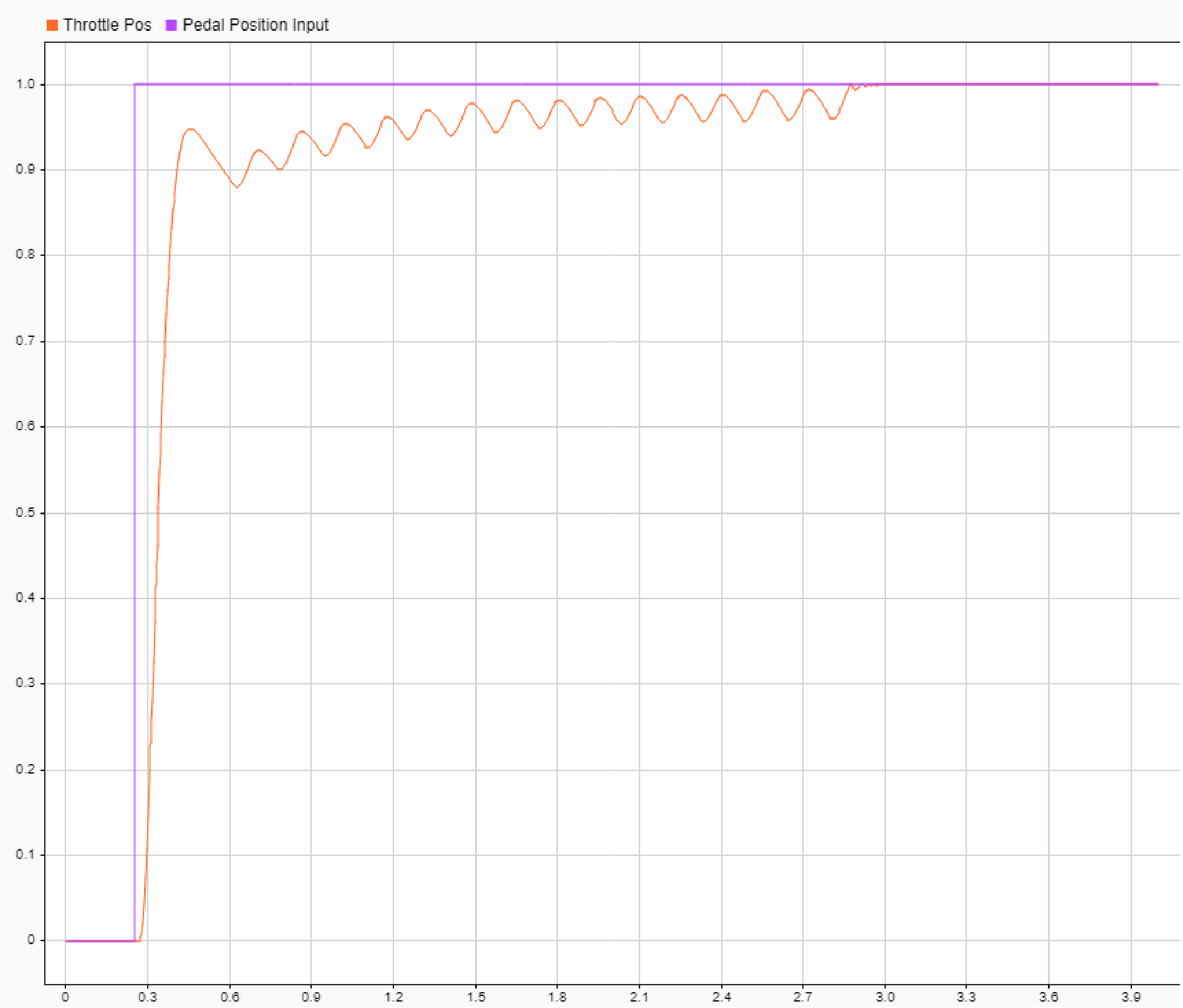
To simulate the throttle control system with the throttle body, use a Model block to reference the software architecture model in the system model. The `ThrottleControlSystem` model also contains a Stateflow® Chart block to model a more complex scheduling of the functions of the software architecture.

```
open_system('ThrottleControlSystem');
```



To simulate the system model containing the plant and Stateflow scheduler, use the command:

```
sim('ThrottleControlSystem');
```

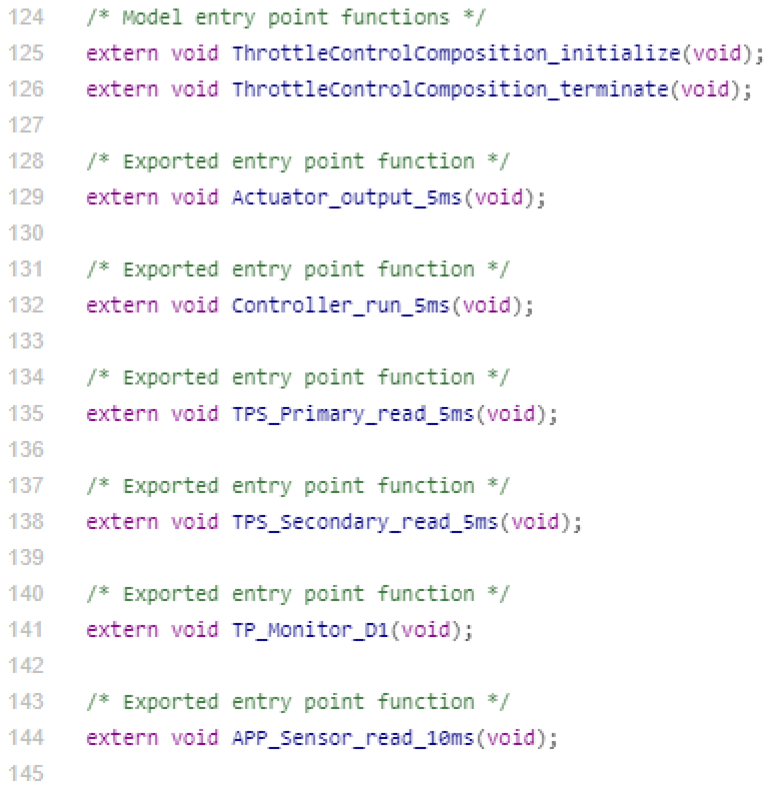


Code Generation

After simulation, you can generate code to deploy the control system to the target hardware. Code generation requires an Embedded Coder® license. Open the `ThrottleControlComposition` model and execute the `slbuild` command, or press **Ctrl+B** to build the model and generate code.

```
slbuild('ThrottleControlComposition');
```

The generated code contains an entry-point function for each function of the components in the software architecture. For more information on code generation for export-function models, see Generate Code for Export-Function Model.



```
/* Model entry point functions */
extern void ThrottleControlComposition_initialize(void);
extern void ThrottleControlComposition_terminate(void);

/* Exported entry point function */
extern void Actuator_output_5ms(void);

/* Exported entry point function */
extern void Controller_run_5ms(void);

/* Exported entry point function */
extern void TPS_Primary_read_5ms(void);

/* Exported entry point function */
extern void TPS_Secondary_read_5ms(void);

/* Exported entry point function */
extern void TP_Monitor_D1(void);

/* Exported entry point function */
extern void APP_Sensor_read_10ms(void);

```