

System Composer™

User's Guide



MATLAB® & SIMULINK®

R2022a



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

© COPYRIGHT 2019–2022 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

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

1	Architecture Model Editing	
	Compose Architecture Visually	1-2
	Create Architecture Model	1-2
	Components	1-5
	Ports	1-9
	Connections	1-12
	Decompose and Reuse Components	1-17
	Decompose Component	1-17
	Create Reference Architecture	1-18
	Use Reference Architecture	1-20
Remove Reference Architecture	1-20	
Create Variants	1-21	
Add Variant Choices	1-22	
Create Software Architecture from Component	1-23	
Build Architecture Models Programmatically	1-24	
Modeling System Architecture of Small UAV	1-32	
2	Requirements	
	Link and Trace Requirements	2-2
	Manage Requirements	2-8
	Mobile Robot Architecture Model	2-8
	Manage Requirements	2-8
	Trace Requirements	2-9
	Use Requirements Traceability Diagram	2-10
	Link Requirements	2-11
	Verify and Validate Requirements Using Test Harnesses	2-13
	Update Reference Requirement Links from Imported File	2-15
3	Interface Management	
	Define Port Interfaces Between Components	3-2

Create Interfaces	3-5
Mobile Robot Architecture Model	3-5
Open Interface Editor	3-5
Create Composite Data Interfaces	3-6
Create Value Types as Interfaces	3-7
Nest Interfaces to Reuse Data	3-8
Assign Interfaces to Ports	3-10
Mobile Robot Architecture Model with Interfaces	3-10
Associate Ports with Interfaces in Property Inspector	3-10
Define Owned Interfaces Local to Ports	3-11
Select Multiple Ports and Assign Data Interface	3-13
Specify Source Element or Destination Element for Ports	3-14
Interface Adapter	3-16
Manage Interfaces with Data Dictionaries	3-20
Mobile Robot Architecture Model with Interfaces	3-20
Save, Link, and Delete Interfaces	3-20
Reference Data Dictionaries	3-23
Add Referenced Data Dictionaries	3-23
Use Referenced Data Dictionaries for Projects with Multiple Models	3-24

Define Architectural Properties

4

Define Profiles and Stereotypes	4-2
Create a Profile and Add Stereotypes	4-2
Add Properties with Stereotypes	4-3
Define Default Stereotypes	4-5
Use Stereotype-Based Styling	4-7
Use Stereotypes and Profiles	4-10
Import Profiles	4-10
Apply Stereotypes	4-12
Remove Stereotypes	4-19
Extend Stereotypes	4-19
Simulate Mobile Robot with System Composer Workflow	4-22
Organize and Link Requirements	4-24
Link Stakeholder Requirements to System Requirements	4-24
Design Architecture Models	4-27
Design, Specify, and Allocate Architecture Models	4-27
Define Stereotypes and Perform Analysis	4-34
Define Stereotypes and Perform Analysis	4-34
Simulate Architectural Behavior	4-43
Simulate Architectural Behavior	4-43

Describe Component Behavior Using Simulink	5-2
Create Simulink Behavior with Robot Arm Model	5-2
Create Referenced Simulink Behavior Model	5-4
Create Simulink Subsystem Behavior Using Subsystem Component	5-6
Link to Existing Simulink Behavior Model	5-9
Access Model Arguments as Parameters on Reference Components	5-9
Create Simulink Behavior from Template for Component	5-10
Use Parameters to Store Instance Values with Components	5-13
Extract Architecture of Simulink Model Using System Composer	5-17
Describe Component Behavior Using Stateflow Charts	5-23
Add State Chart Behavior to Component	5-23
Remove Stateflow Chart Behavior from Component	5-26
Extract Architecture from Simulink Model	5-28
Describe System Behavior Using Sequence Diagrams	5-32
Traffic Light Example for Sequence Diagrams	5-32
Author Sequence Diagram for Traffic Light Example	5-38
Traffic Light Example	5-39
Add Lifelines and Messages	5-40
Add Fragments and Operands	5-45
Use Sequence Diagrams with Architecture Models	5-51
Traffic Light Example with Hierarchy for Sequence Diagrams	5-51
Create Sequence Diagram	5-53
Add Child Lifelines to Sequence Diagram	5-55
Create Sequence Diagram Gates	5-57
Co-Create Components	5-58
Synchronize Sequence Diagram and Model	5-59
Create Messages in Sequence Diagram	5-59
Modify Sequence Diagram Using Model Browser	5-62
Create Sequence Diagram from View	5-63
Implement Sequence Diagram Fragments	5-65
Sequence Diagram Fragments	5-65
Fragment Semantics	5-66
Describe Component Behavior Using Simscape	5-75
Architecture Model with Simscape Behavior for a DC Motor	5-75
Define Physical Ports on Component	5-76
Specify Physical Interfaces on Ports	5-76
Create Simulink Subsystem Component	5-77
Describe Component Behavior Using Simscape	5-78
Synchronize Sequence Diagrams and Architecture Models	5-81

Allocate Architecture Models

6

Create and Manage Allocations	6-2
Allocate Architectures in Tire Pressure Monitoring System	6-5
Systems Engineering Approach for SoC Applications	6-10

Analyze Architecture Model

7

Analyze Architecture	7-2
Set Properties for Analysis	7-2
Create a Model Instance for Analysis	7-4
Write Analysis Function	7-6
Run Analysis Function	7-7
Analysis Function Constructs	7-9
Roll-Up Analysis for Quadcopter Design	7-9
Class-Based Analysis for Battery Sizing	7-10
Allocation-Based Analysis for Tire Pressure Monitoring	7-11
Remaining Useful Life Analysis for Mobile Robot Design	7-11
Variant Analysis for Insulin Infusion Pump Design	7-12
Battery Sizing and Automotive Electrical System Analysis	7-14
Calculate Endurance Using Quadcopter Architectural Design	7-16
Design Insulin Infusion Pump Using Model-Based Systems Engineering	7-23

Software Architectures

8

Author Software Architectures	8-2
Create New Software Architecture Model	8-2
Build a Simple Software Architecture Model	8-3
Import and Export Software Architectures	8-5
Create Software Architecture from Architecture Model Component	8-5
Simulate and Deploy Software Architectures	8-8
Modeling the Software Architecture of a Throttle Position Control System	8-14
Class Diagram View of Software Architectures	8-20
Software Architecture with Class Diagram View	8-20

Interact with Class Diagram View	8-20
Client-Server Interfaces in the Class Diagram View	8-21
Author and Extend Functions for Software Architectures	8-24
Author and Visualize Functions Using Functions Editor	8-24
Author Functions Programmatically	8-28
Implement Behaviors for Functions in the Architecture Level	8-29
Apply Stereotypes to Functions of Software Architectures	8-29
Import and Export Functions of Software Architectures	8-30
Merge Message Lines Using Adapter Block	8-32
Authoring Functions for Software Components of an Adaptive Cruise Control	8-34
Author Service Interfaces for Client-Server Communication	8-41
Service-Oriented Sensor Modeling	8-44

Create Custom Views

9

Create Spotlight Views	9-2
Mobile Robot Architecture Model with Properties	9-2
Create Spotlight Views from Components	9-2
Create Architecture Views Interactively	9-5
Create Filtered Views with Component Filters and Port Filters	9-5
Add Group By Criteria to Filtered Views	9-9
Interactively Add and Remove Elements from Views	9-10
Add or Remove Requirements Links from Views	9-11
Add Custom Clauses to Component Filters and Port Filters	9-13
Create Architectural Views Programmatically	9-15
Create Architecture Views in System Composer with Keyless Entry System	9-15
Find Elements in Model Using Queries	9-17
Display Component Hierarchy and Architecture Hierarchy Using Views	9-21
Robot Computer Systems Architecture	9-21
Switch Between Component Diagram View and Hierarchy Views	9-22
Modeling System Architecture of Keyless Entry System	9-25

Manage Architecture Models

10

Organize System Composer Files in Projects	10-2
Use Projects to Organize Files and Folders	10-2
Compare Model Differences Using System Composer Comparison Tool	10-4

Import and Export Architecture Models

11

Import and Export Architectures	11-2
Import and Export Architecture Models	11-5
Define Basic Architecture	11-5
Import Basic Architecture	11-6
Extend Basic Architecture Import	11-7
Export Architecture	11-11
Import System Composer Architecture Using ModelBuilder	11-13

Architecture Model Editing

- “Compose Architecture Visually” on page 1-2
- “Decompose and Reuse Components” on page 1-17
- “Build Architecture Models Programmatically” on page 1-24
- “Modeling System Architecture of Small UAV” on page 1-32

Compose Architecture Visually

In this section...
“Create Architecture Model” on page 1-2
“Components” on page 1-5
“Ports” on page 1-9
“Connections” on page 1-12

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

With MATLAB® code and the functions `importModel` and `exportModel`, you can import external architecture descriptions into System Composer. For more information, see “Import and Export Architecture Models” on page 11-5.

Alternatively, you can use MATLAB programming to create and customize the various architectural elements. For details, see “Build Architecture Models Programmatically” on page 1-24.

Create Architecture Model

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 file that contains architectural information, including components, ports, connectors, interfaces, and behaviors.

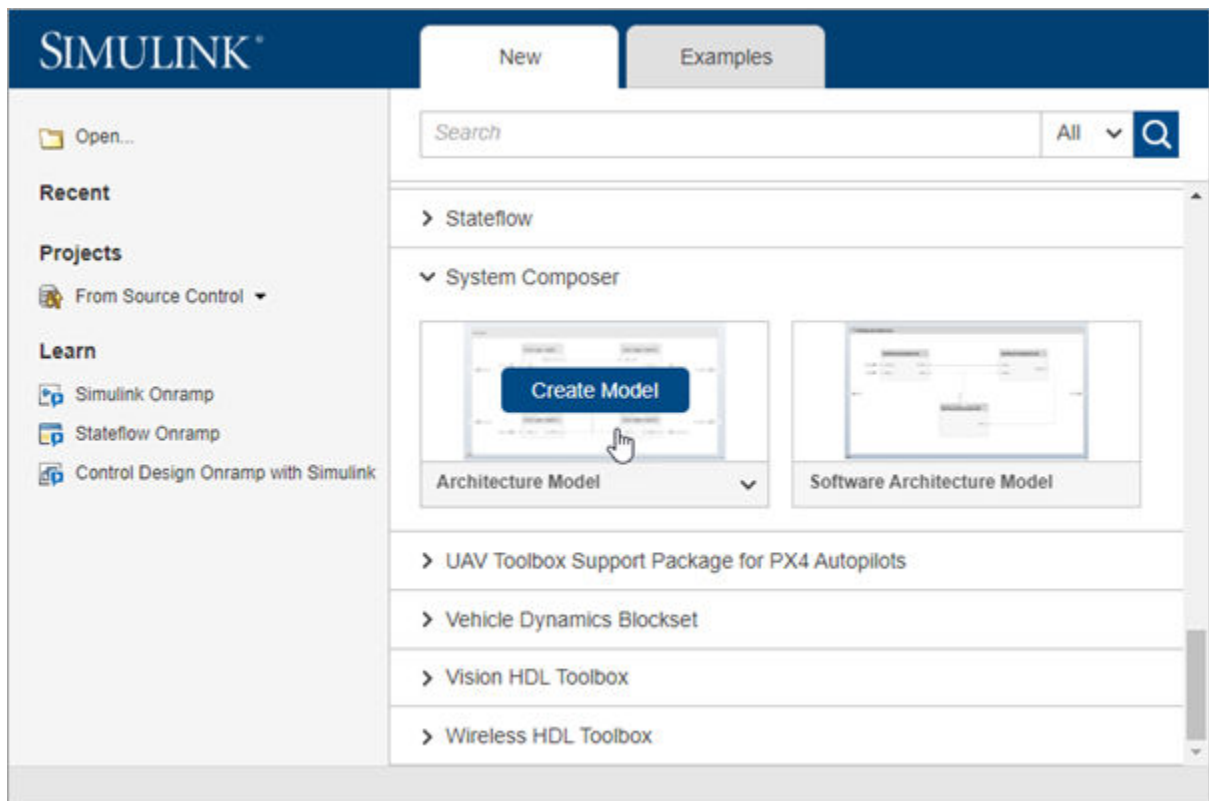
An architecture model includes a top-level architecture that holds the composition of the system. This top-level architecture also allows definition of interfaces of this system with other systems.



Start with a blank architecture model to model the physical and logical architecture of a system. Use one of these three methods to create an architecture model:

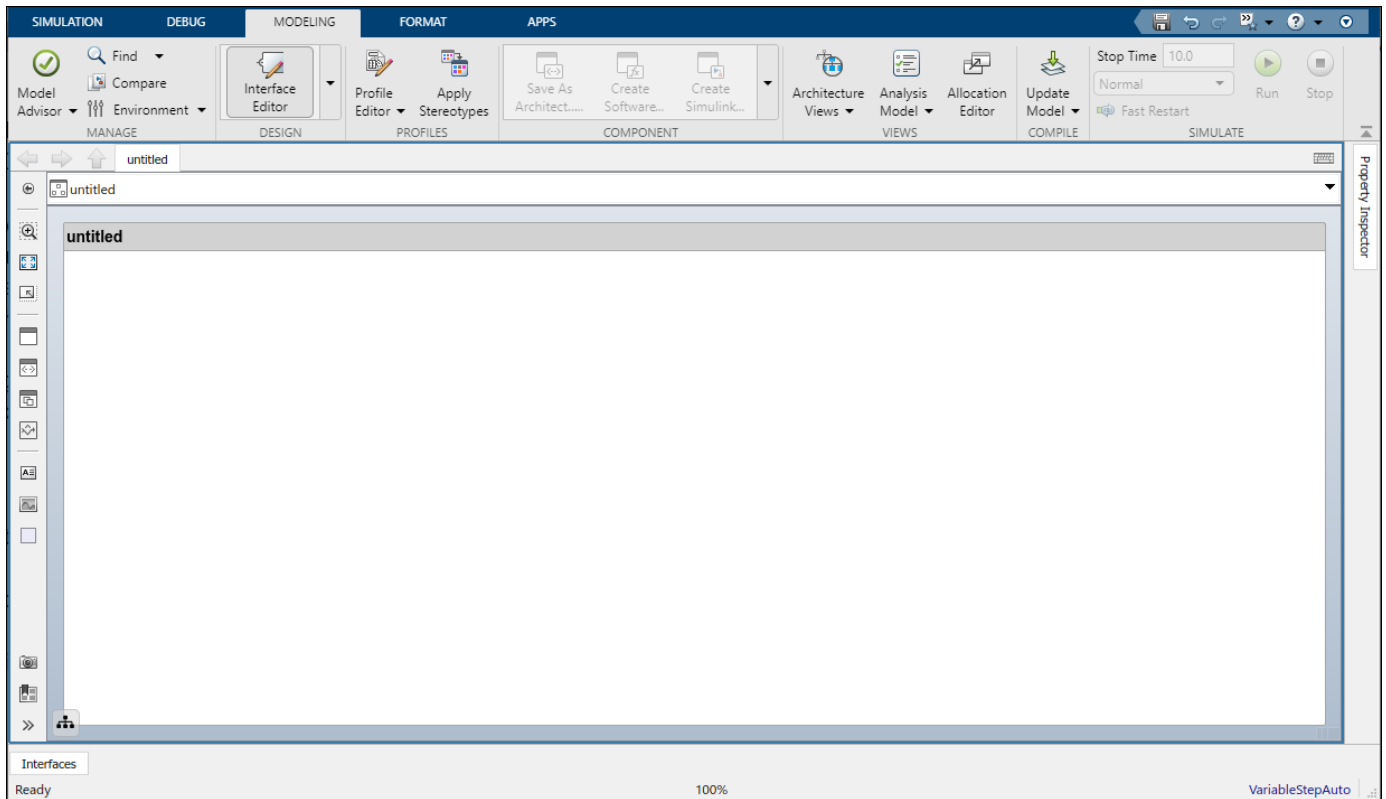
- At the MATLAB Command Window, enter:

```
systemcomposer
```

Select **Architecture Model**.




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



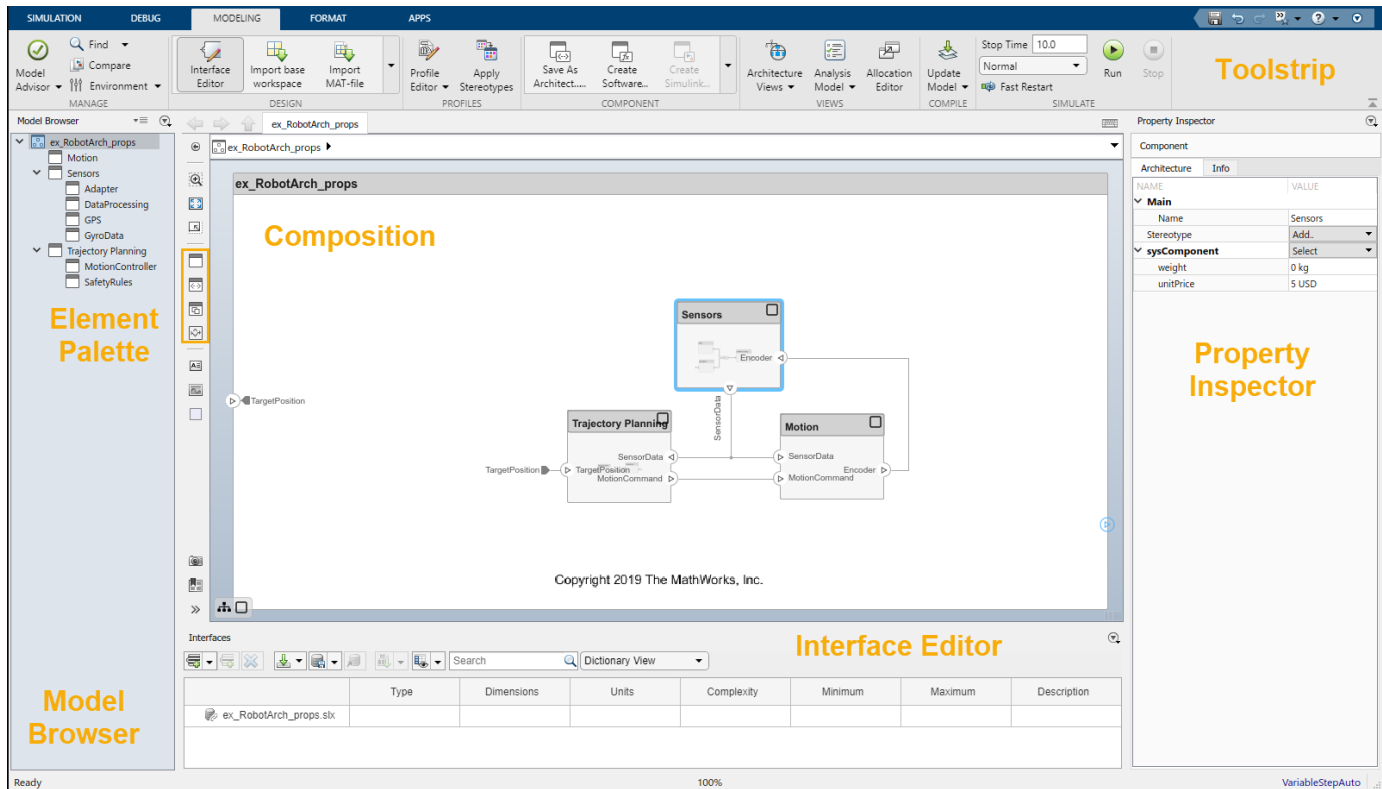
- At the MATLAB Command Window, enter:

```
archModel = systemcomposer.createModel("ModelName");  
systemcomposer.openModel(archModel);
```

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

Save the architecture model. On the **Simulation** tab, select **Save** . 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 architectural 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.

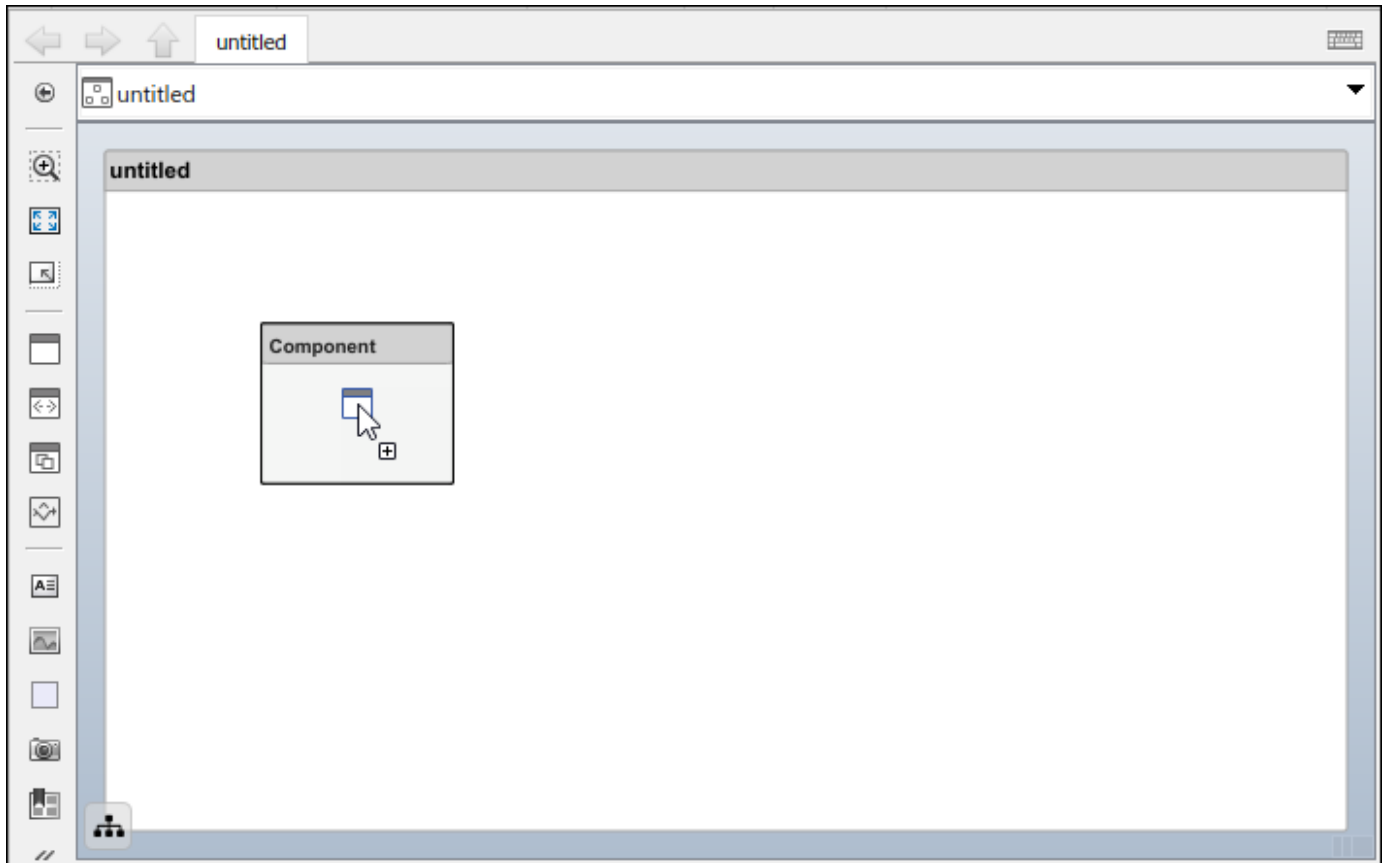
Represented as a block, a component is a part of an architecture model that can be separated into reusable artifacts.

Add Components

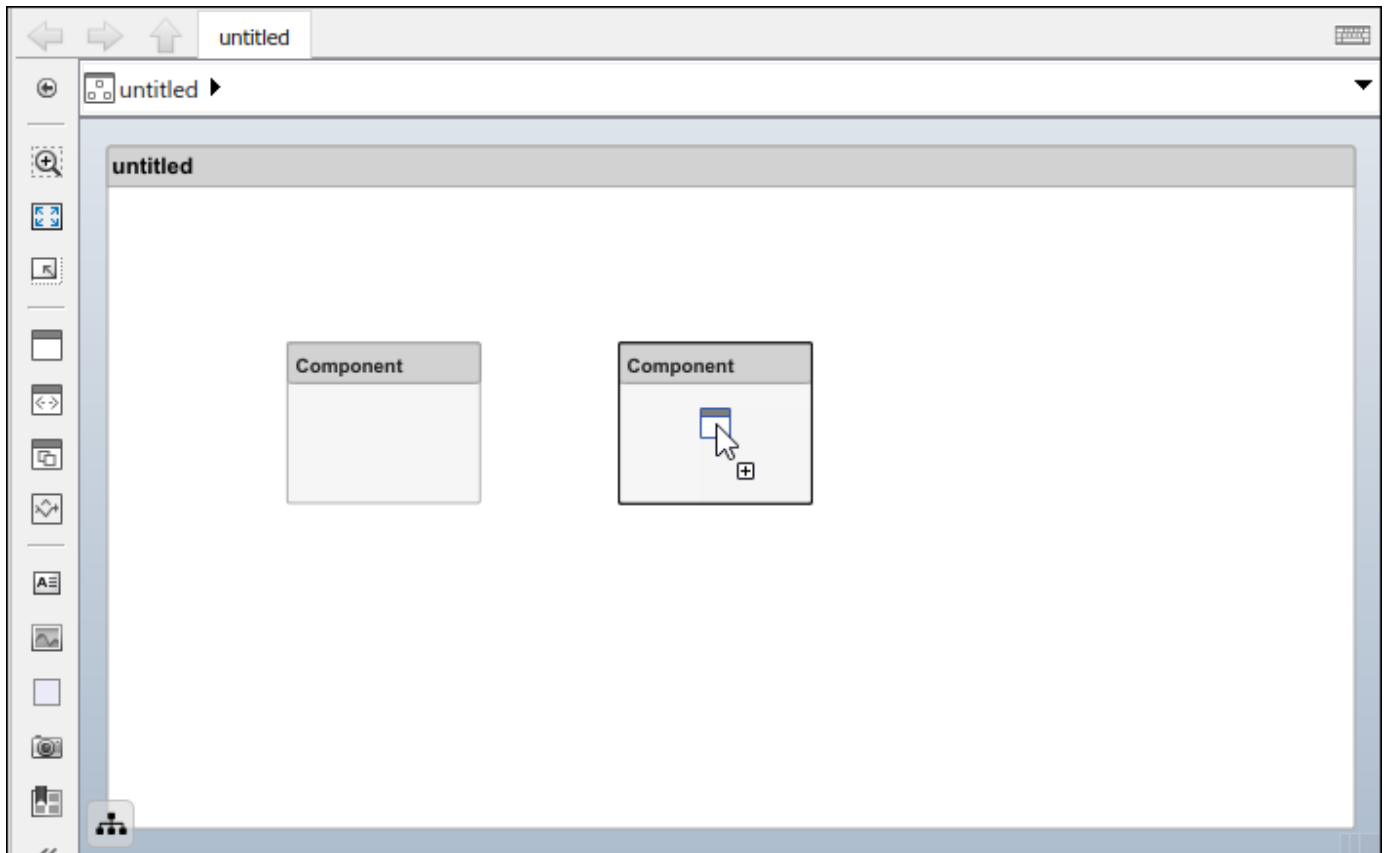
Use one of these methods to add components to the architecture:

- 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 —

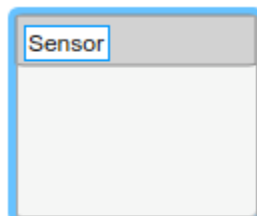
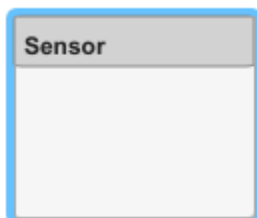


- Create multiple components from the palette —



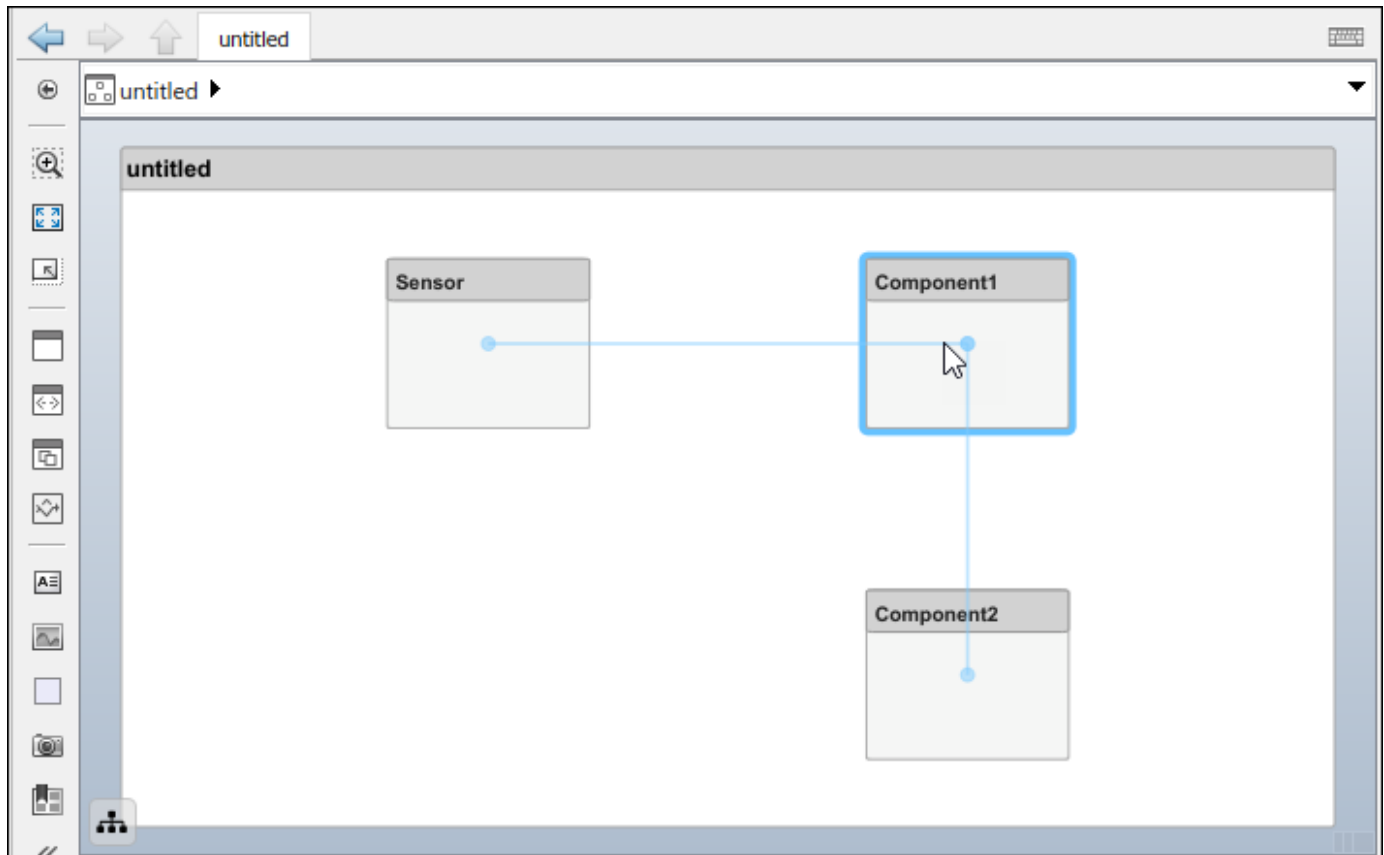
Name Component

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 Component

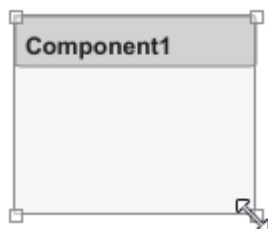
Move a component simply by clicking and dragging it. Blue guidelines may appear to help align the component with other components.



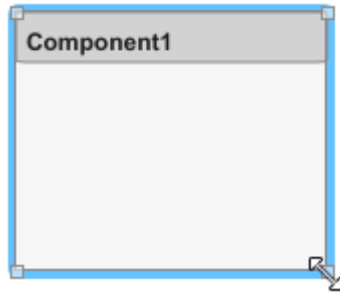
Resize Component

Resize a component by dragging corners.

- 1 Pause the pointer over a corner to see the double arrow.



- 2 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 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**.

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:

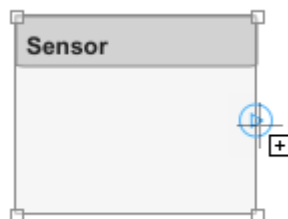
- *Component ports* are interaction points on the component to other components.
- *Architecture ports* are ports on the boundary of the system, whether the boundary is within a component or the overall architecture model.

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 Port Interfaces Between Components” on page 3-2.

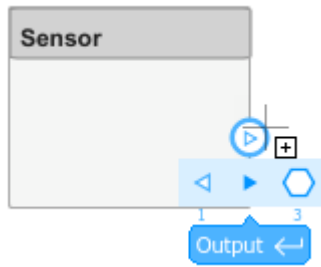
Add Component Port

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.

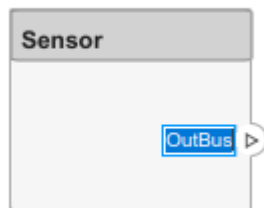
- 1 Pause over the side of a component. A + sign and a port outline appear.






- 2 Click the port outline. A set of options appear for an Input, Output, or Physical port.



- 3 Select Output to commit the port. You can also name the port at this point.

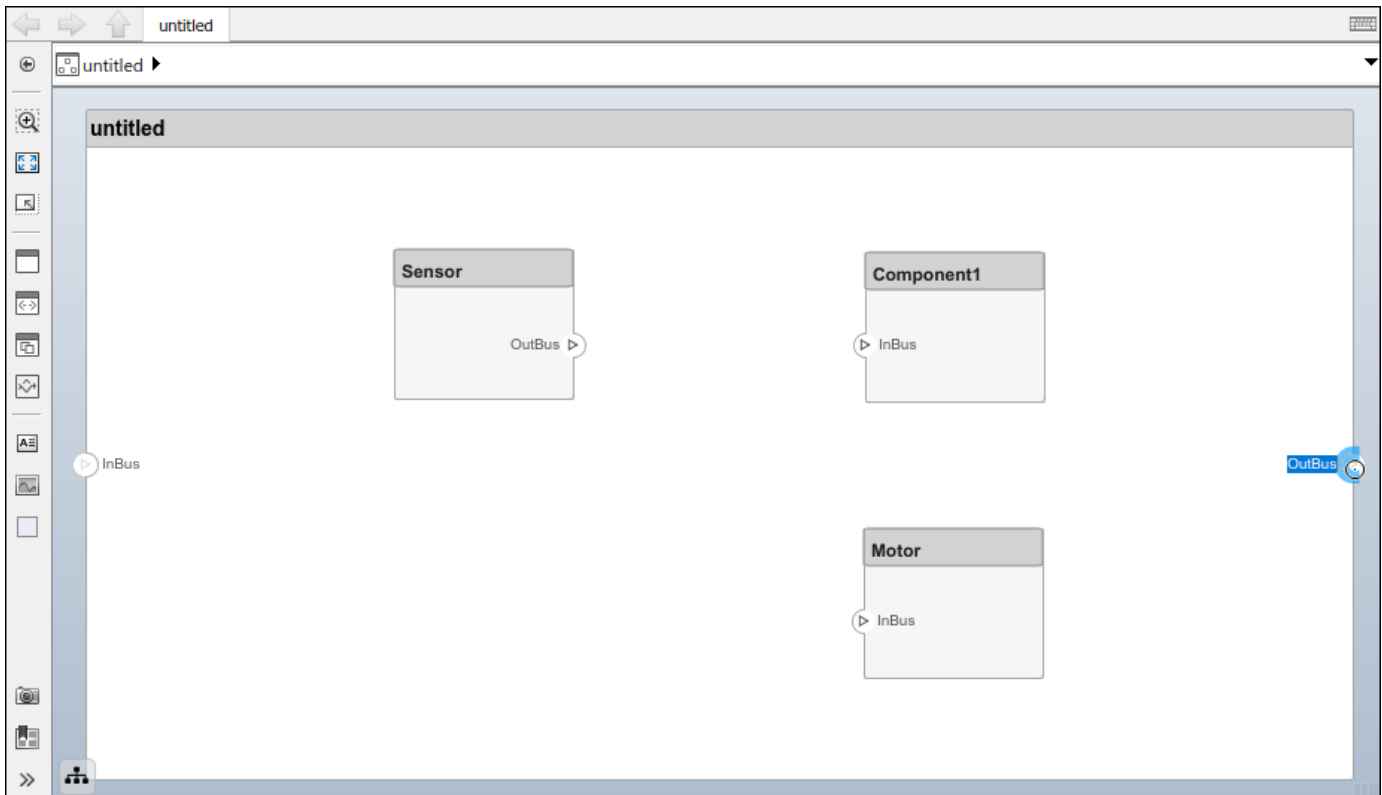


An output port is shown with the  icon, an input port is shown with the  icon, and a physical port is shown with the  icon and is nondirectional.

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

Add 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 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 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

Arrow Key	Original Port Edge	Port Movement
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 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.

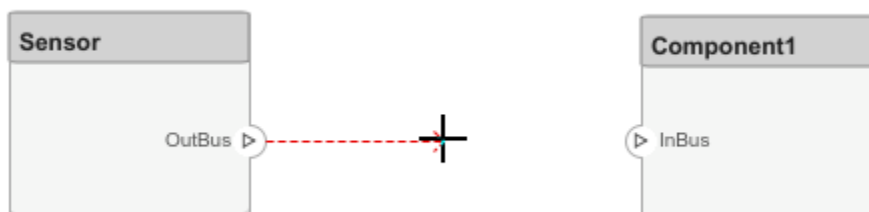
A connector allows two components to interact without defining the nature of the interaction. Set an interface on a port to define how the components interact.

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

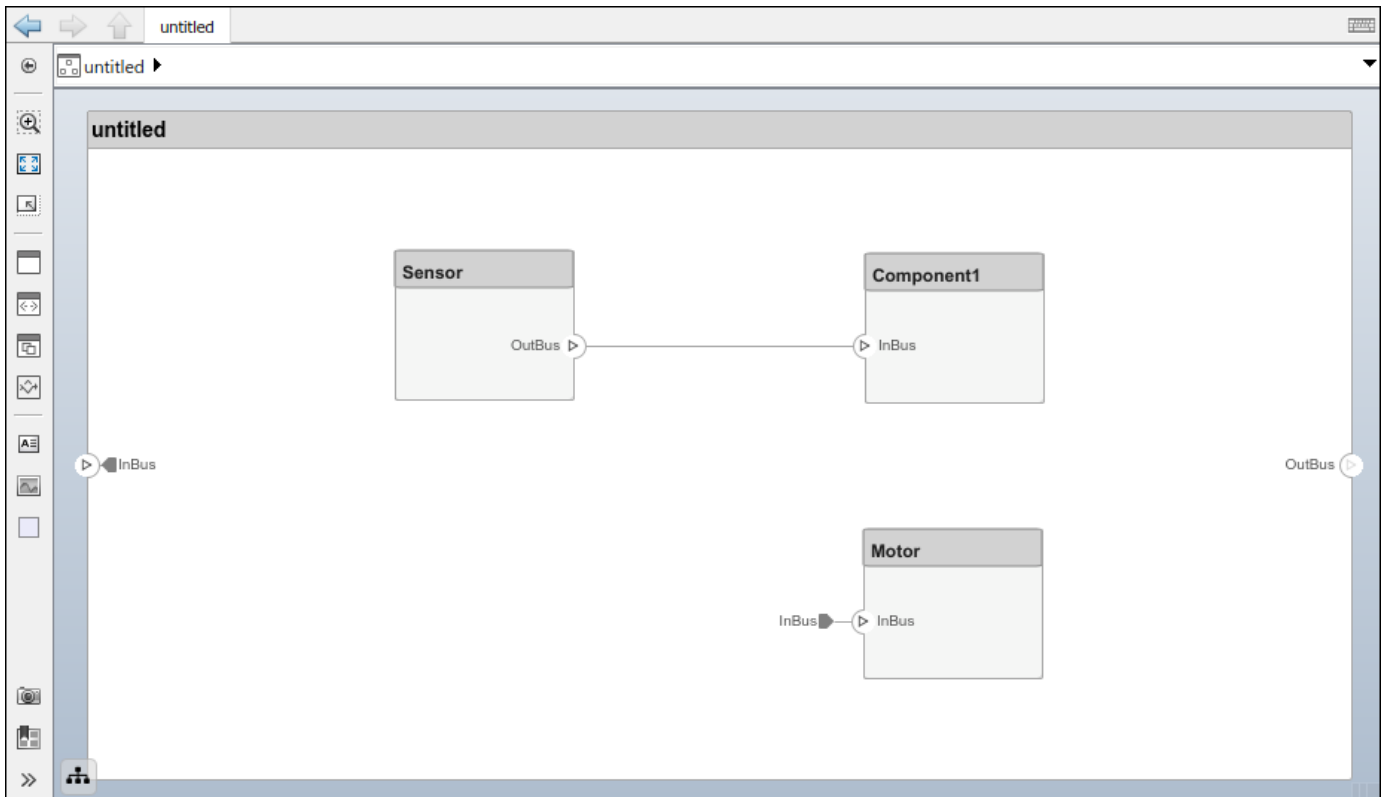
Connect two ports by dragging a line:

- 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.



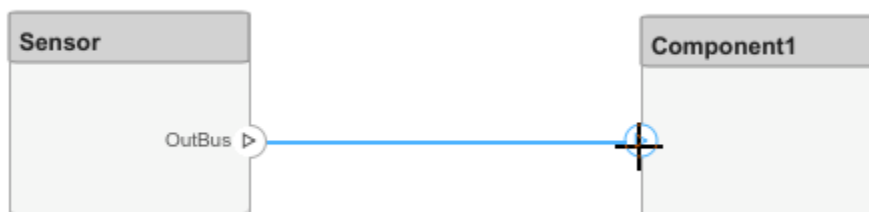
You can take these steps in both directions — input port to output port, or output port to input port. You cannot connect ports that have the same direction.

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



Connect Components Without Ports

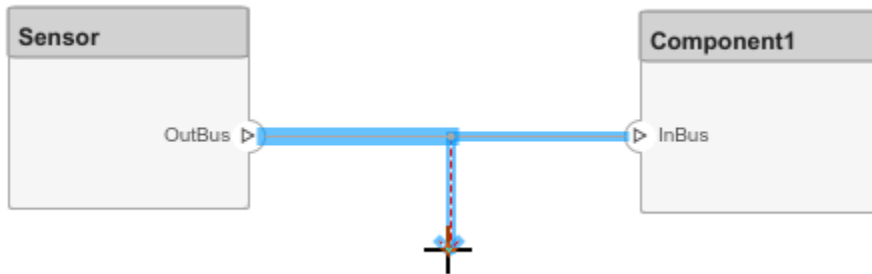
To quickly create ports and connections at the same time, drag a line from one component edge to another. The direction of this connection depends on which edges of the components are used - left and top edges are considered inputs, right and bottom edges are considered outputs. You can also perform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

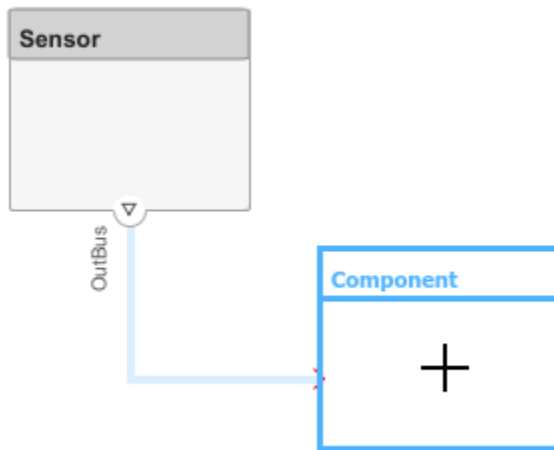
Branch Connections

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.



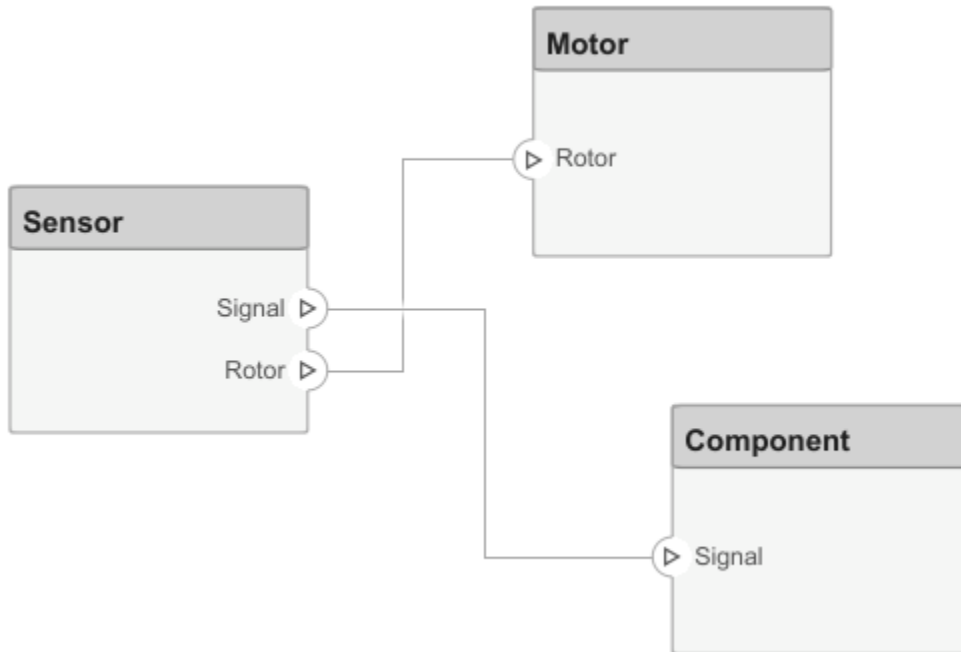
Create New Components Through Connections

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.

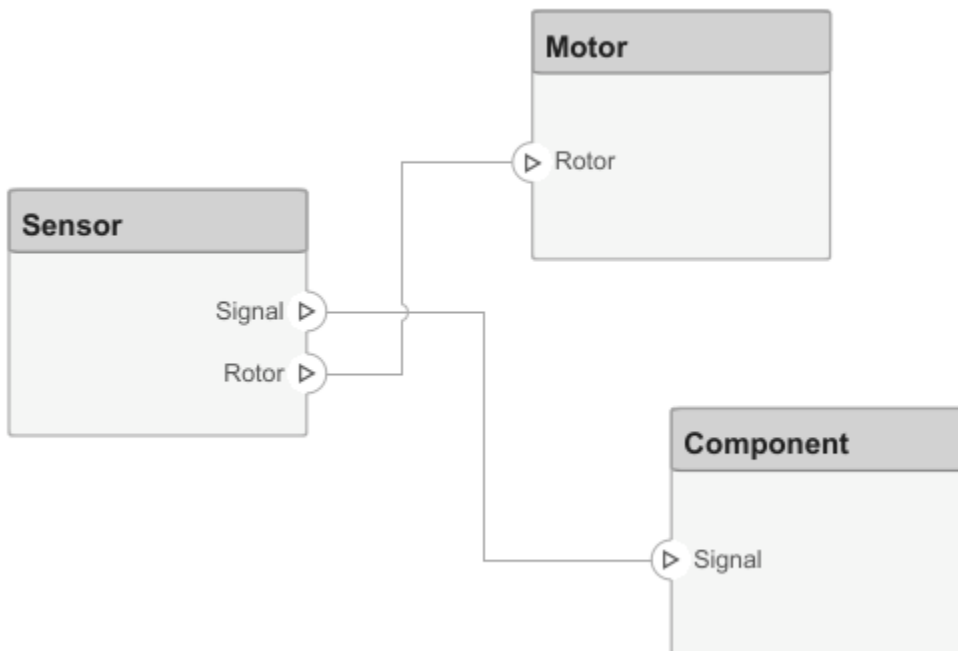


Change Line Crossing Style for Overlapping Connections

In complex architectural diagrams, connectors can overlap. You can improve the readability of your diagram by choosing another line crossing style. Navigate to **Modeling > Environment > Simulink Preferences**. In **Simulink Preferences**, select **Editor**, then select a **Line crossing style**. The default line crossing style, Tunnel, is shown below.



Another option, Line Hop, is shown below.



For more information on line crossing style parameters, see "Line crossing style".

See Also

Functions

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

Blocks

Component

More About

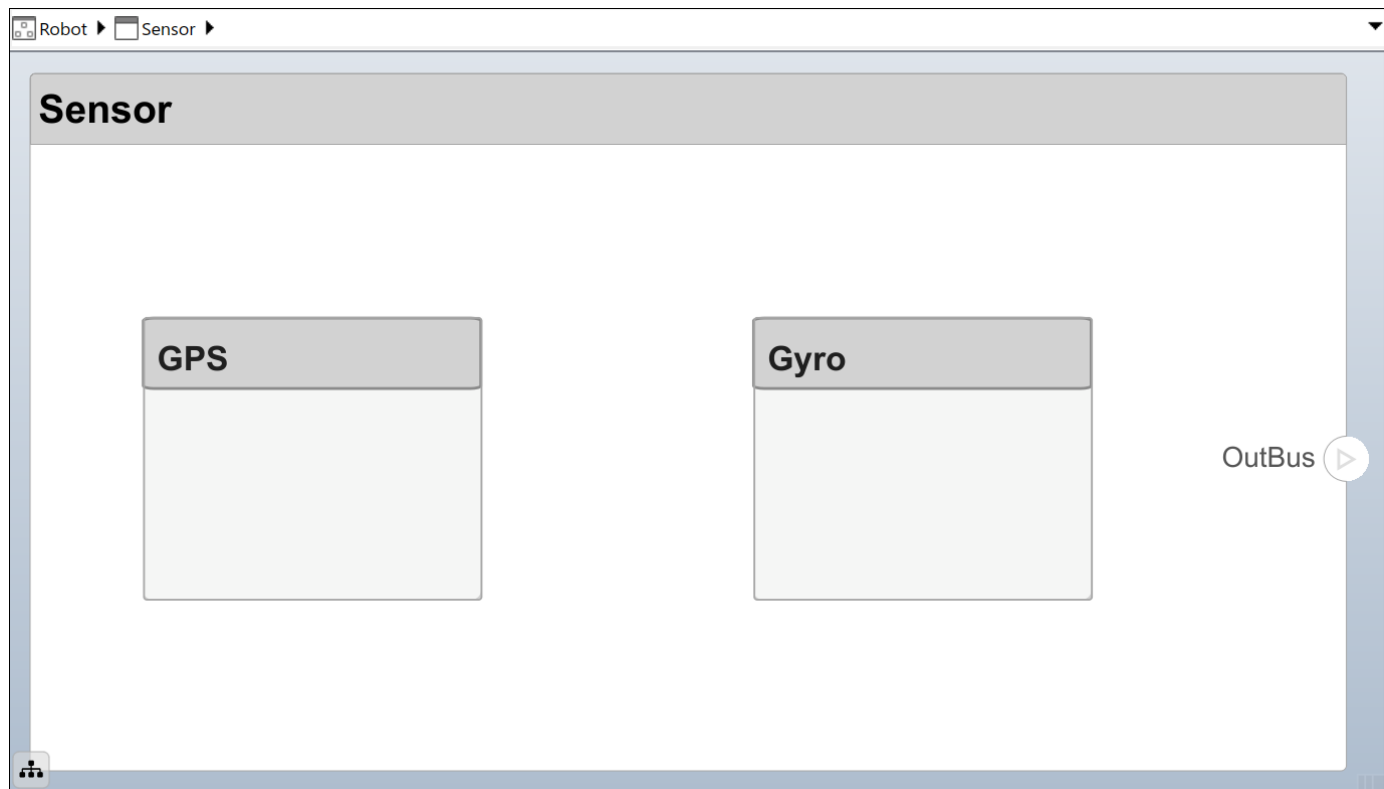
- “Decompose and Reuse Components” on page 1-17
- “Create Interfaces” on page 3-5
- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Organize System Composer Files in Projects” on page 10-2
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

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 Component

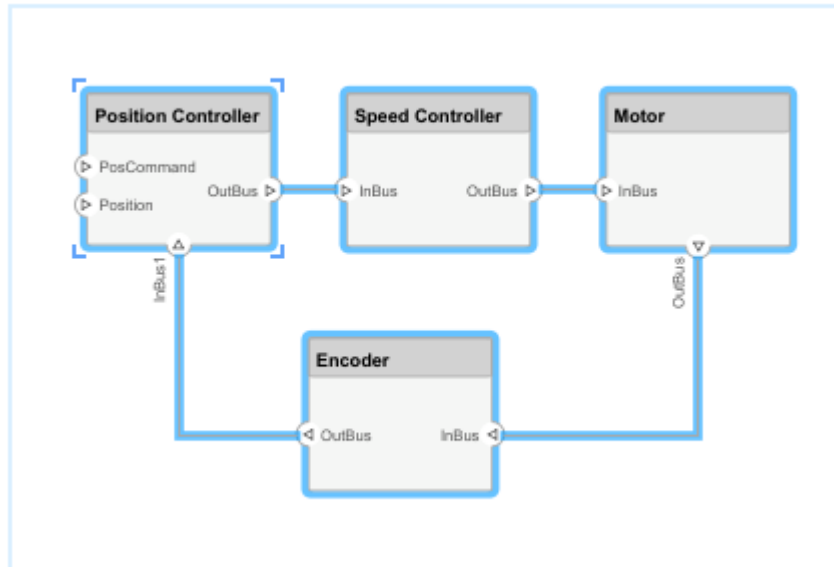
A component can have its own architecture. Double-click a component to view or edit its architecture. When you view the component at this level, its ports appear as architecture ports. Use the Model Browser to view component hierarchy.



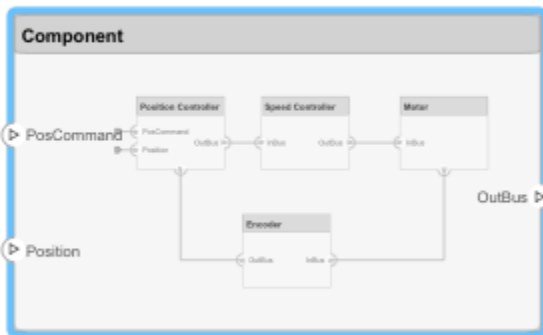
You can add components, ports, and connections at this level to define the architecture.

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 by right-clicking and selecting Create Component from Selection.



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.

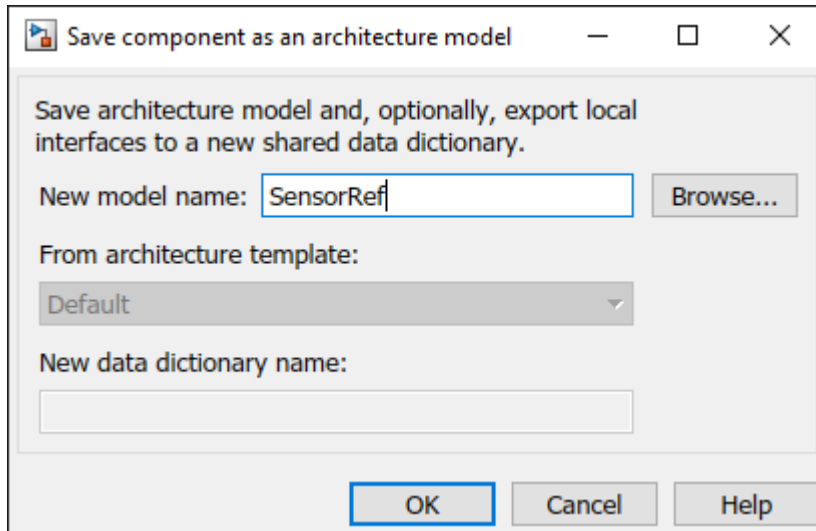
Any component that has its own architecture displays a preview of its contents.

Create Reference Architecture

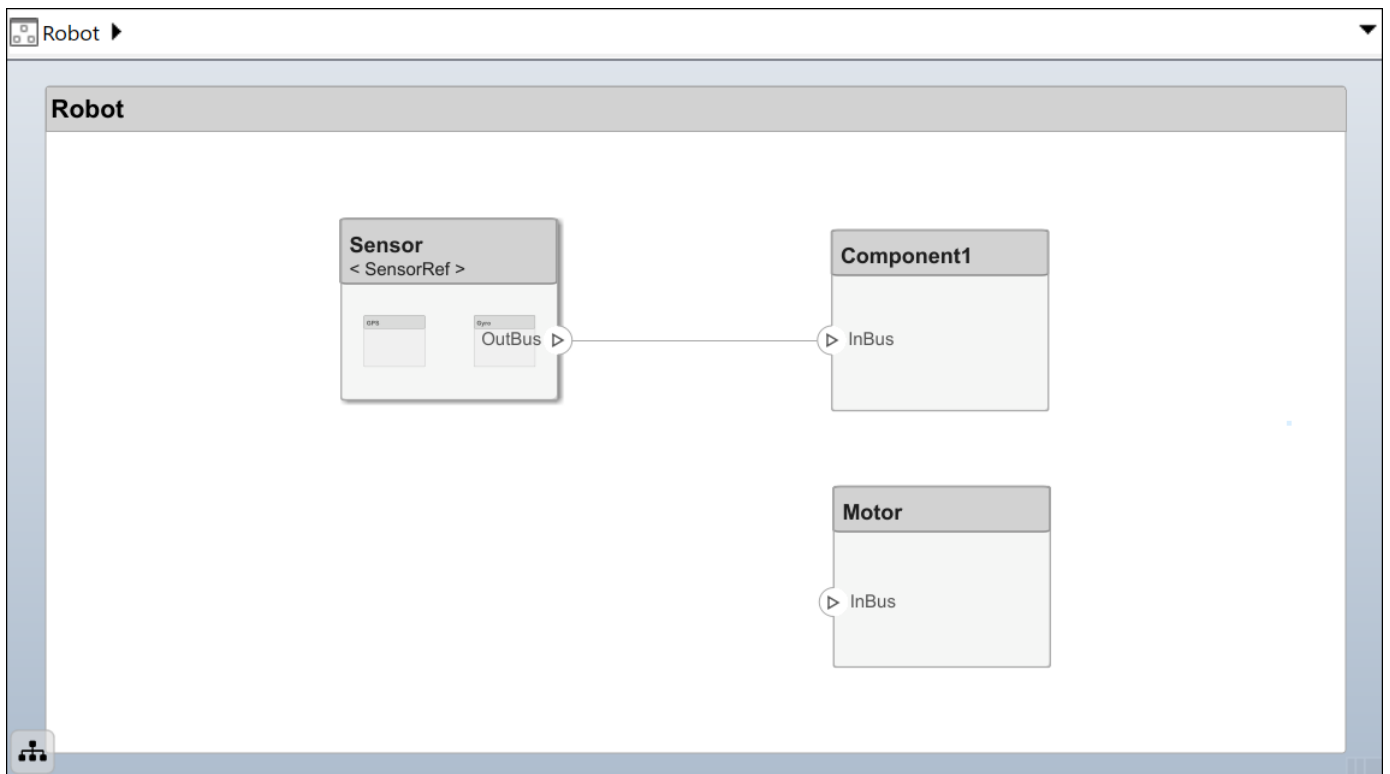
Some projects use the same, detailed component in multiple places, and require the design of such a component to be tightly managed. You can create a reference architecture to reuse the architectural definition of a component in the same architecture model or across several architecture models. Create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

- 1 Right-click the Sensor 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.



- 3 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.



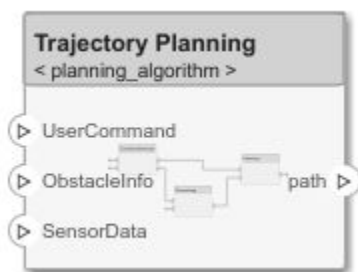
All architecture models can reference this new architecture model through linked components.

Use 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 Trajectory Planning component and select **Link to Model**.

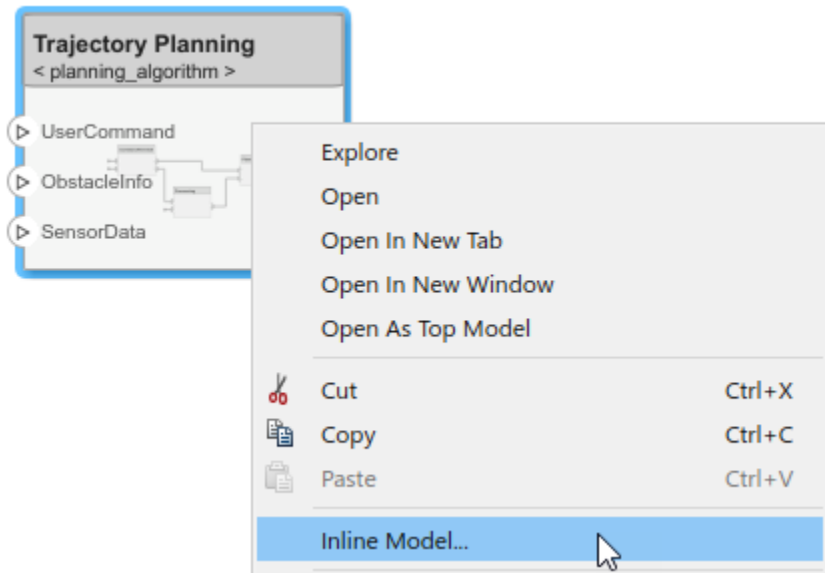
Provide the full path to the reference architecture. If the linked component has its own ports and components, this content is deleted during linking and replaced by that of the reference architecture. The ports of the linked component become the architecture ports in the reference architecture.



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.

Remove 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 remove 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 removed, 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 remove a Stateflow® chart behavior, see “Remove Stateflow Chart Behavior from Component” on page 5-26.

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.

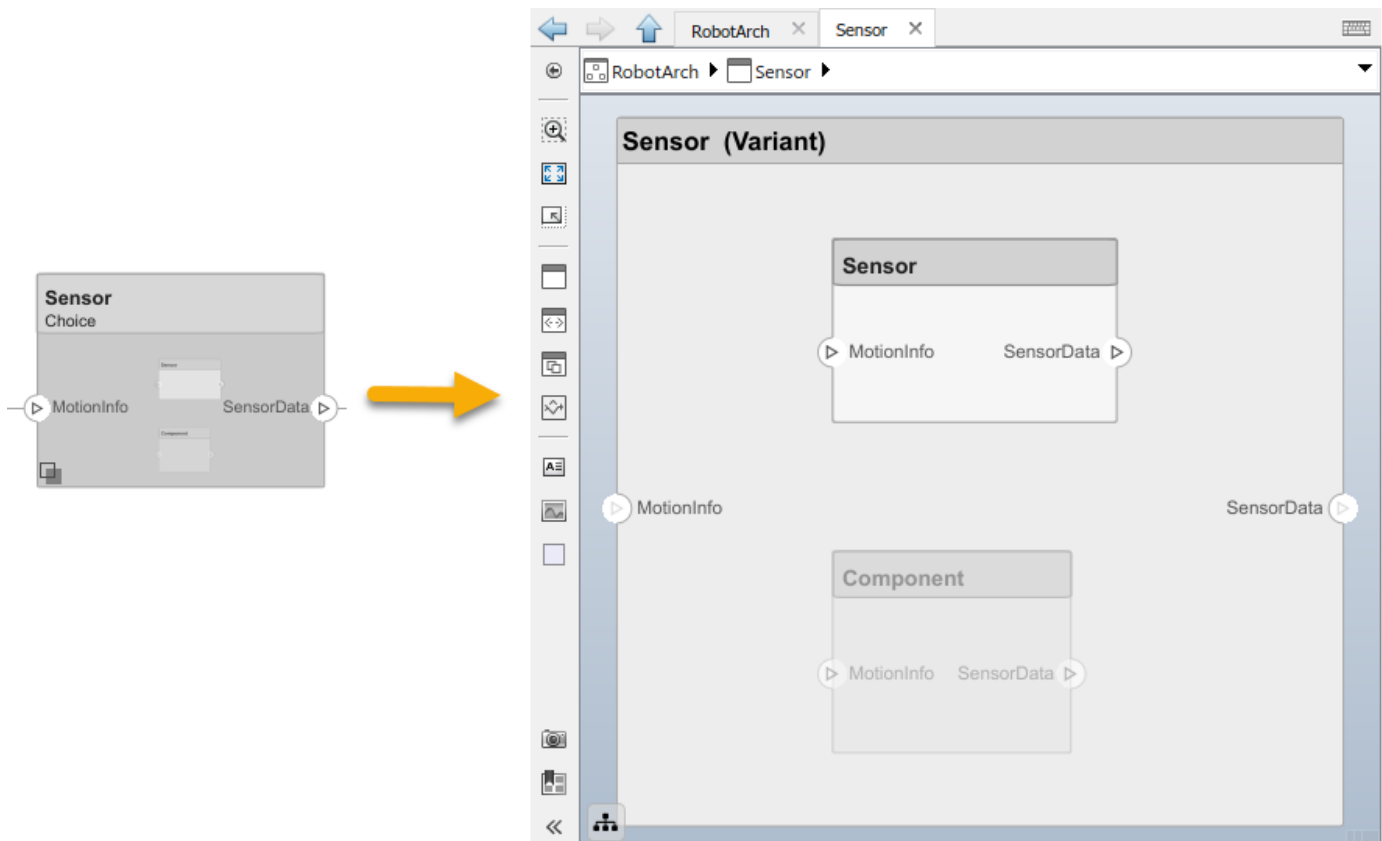
A variant control is a string that controls the active variant choice.

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.

Convert a Component to a Variant Component adding variant choices to the component. Right-click the Sensor 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.



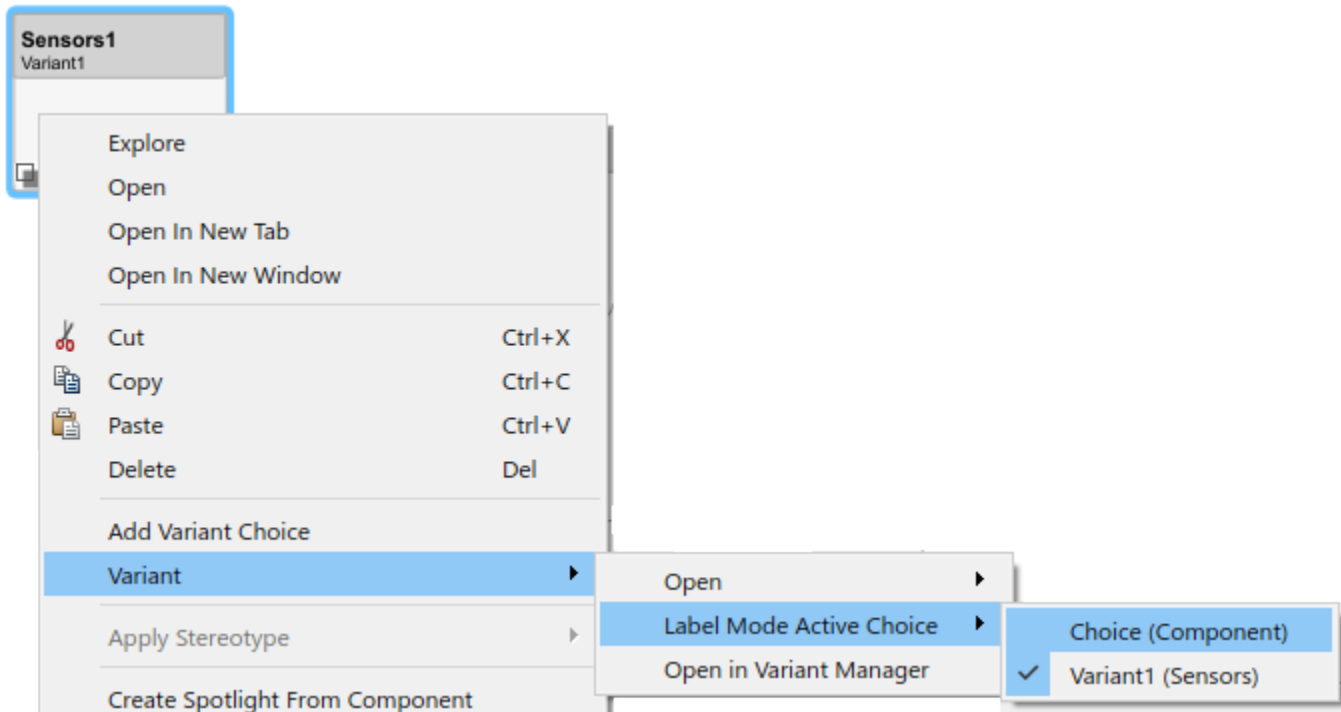
Add Variant Choices

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 > Choice (Component)**. The active choice is displayed in the header of the block.



Create Software Architecture from Component

You can create a software architecture model from a component in a System Composer architecture model and reference the software architecture model from the component. You can use software architectures to link Simulink export-function, rate-based, or JMAAB models to components in your architecture model to simulate and generate code. For more information, see “Create Software Architecture from Architecture Model Component” on page 8-5.

See Also

Functions

`createArchitectureModel` | `linkToModel` | `inlineComponent` | `addVariantComponent` | `makeVariant` | `addChoice` | `setActiveChoice`

Blocks

Reference Component | Variant Component

More About

- “Describe Component Behavior Using Simulink” on page 5-2
- “Describe Component Behavior Using Stateflow Charts” on page 5-23
- “Organize System Composer Files in Projects” on page 10-2
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

Build Architecture Models Programmatically

Build an architecture model programmatically using System Composer™.

Build Model

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

Add Components, Ports, Connections, and Interfaces

Create a model and extract its architecture.

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

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

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

Save the changes to the interface data dictionary.

```
dictionary.save
```

Save the model.

```
model.save
```

Open the model.

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

View the interfaces in the Interface Editor.

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

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

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

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

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

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

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

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

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

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

```

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

```

Add and Connect Architecture Port

Add an architecture port on the architecture.

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

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

```

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

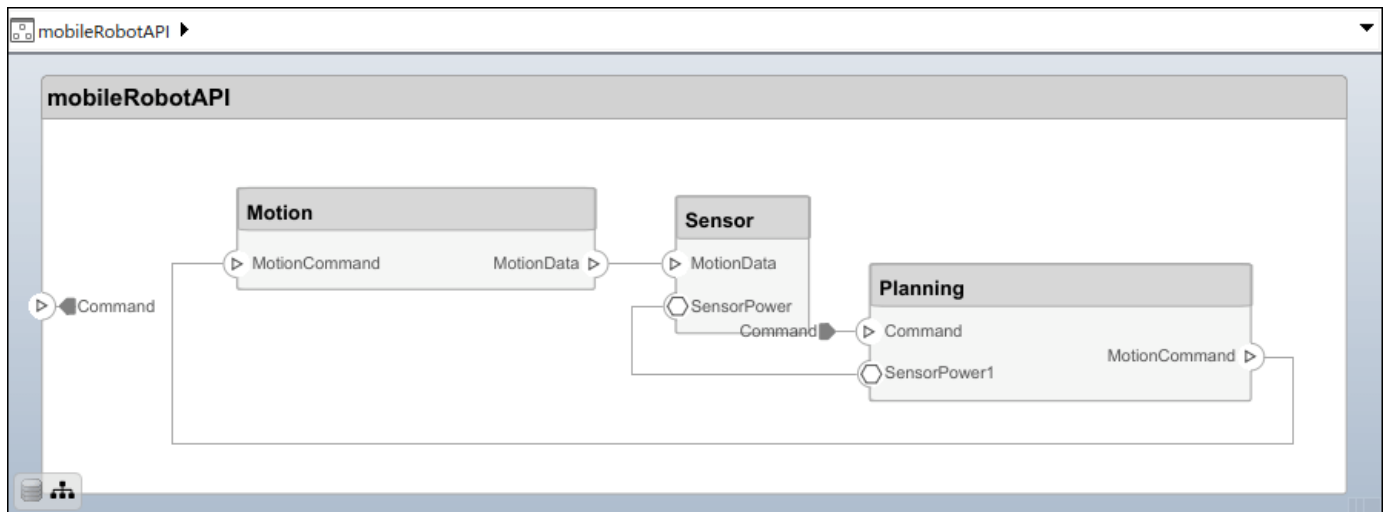
```

Save the model.

```
model.save
```

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

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



Create and Apply Profile with Stereotypes

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

Create Profile and Add Stereotypes

Create a profile.

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

Create a stereotype that applies to all element types.

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

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

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

Create a stereotype for connections.

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

Add Properties

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

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

Save Profile

```
profile.save;
```

Apply Profile to Model

Apply the profile to the model.

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

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

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

Apply the connector stereotype to all connections.

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

Apply the general element stereotype to all connectors and ports.

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

Set properties for each component.

```
setProperty(componentSensor, 'GeneralProfile.projectElement.ID', '001');
setProperty(componentSensor, 'GeneralProfile.projectElement.Description', "'Central unit for all");
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Cost', '200');
setProperty(componentSensor, 'GeneralProfile.physicalComponent.Weight', '450');
setProperty(componentPlanning, 'GeneralProfile.projectElement.ID', '002');
```

```
setProperty(componentPlanning, 'GeneralProfile.projectElement.Description', 'Planning computer');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develCost', '20000');
setProperty(componentPlanning, 'GeneralProfile.softwareComponent.develTime', '300');
setProperty(componentMotion, 'GeneralProfile.projectElement.ID', '003');
setProperty(componentMotion, 'GeneralProfile.projectElement.Description', 'Motor and motor control');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Cost', '4500');
setProperty(componentMotion, 'GeneralProfile.physicalComponent.Weight', '2500');
```

Set the properties of connections to be identical.

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

Add Hierarchy

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

```
motionArch = componentMotion.Architecture;

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

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

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

For output connections, the data element must be specified.

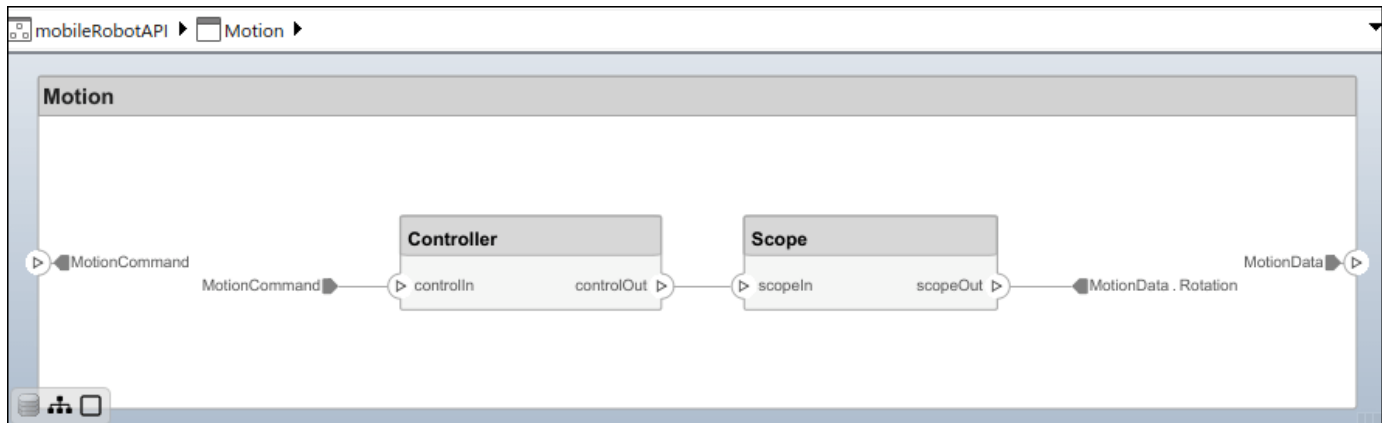
```
c_planningScope = connect(scopeCompPortOut, motionPorts(2), 'DestinationElement', "Rotation");
c_planningConnect = connect(controllerCompPortOut, scopeCompPortIn, 'GeneralProfile.standardConn');
```

Save the model.

```
model.save
```

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

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

Create Model Reference

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

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

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

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



Save the models.

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

Make Variant Component

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

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

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

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

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

Create the necessary ports on `MotionAlt`.

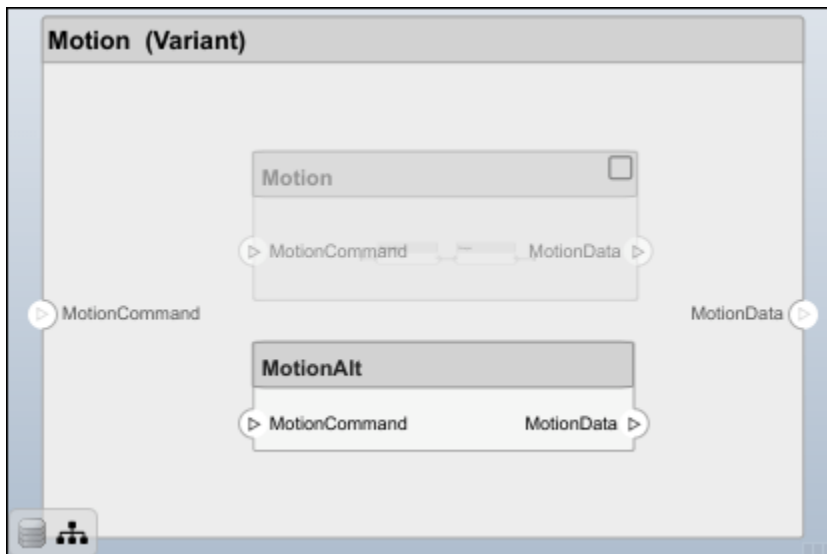
```
motionAltPorts = addPort(choice2.Architecture,{'MotionCommand','MotionData'},{'in','out'});
```

Make `MotionAlt` the active variant.

```
setActiveChoice(variantComp,'MotionAlt')
```

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

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



Save the model.

```
model.save
```

Clean Up

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

```
cleanUpArtifacts
```

See Also

Functions

`createModel` | `createDictionary` | `addInterface` | `addPhysicalInterface` | `addValueType` | `addElement` | `setType` | `createOwnedType` | `linkDictionary` | `addComponent` | `addPort` |

setInterface | connect | save | getPort | createProfile | addStereotype | addProperty |
save | applyProfile | applyStereotype | batchApplyStereotype | setProperty |
linkToModel | makeVariant | addChoice | setActiveChoice | closeAll

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
- “Decompose and Reuse Components” on page 1-17
- “Create Interfaces” on page 3-5
- “Organize System Composer Files in Projects” on page 10-2
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

Modeling System Architecture of Small UAV

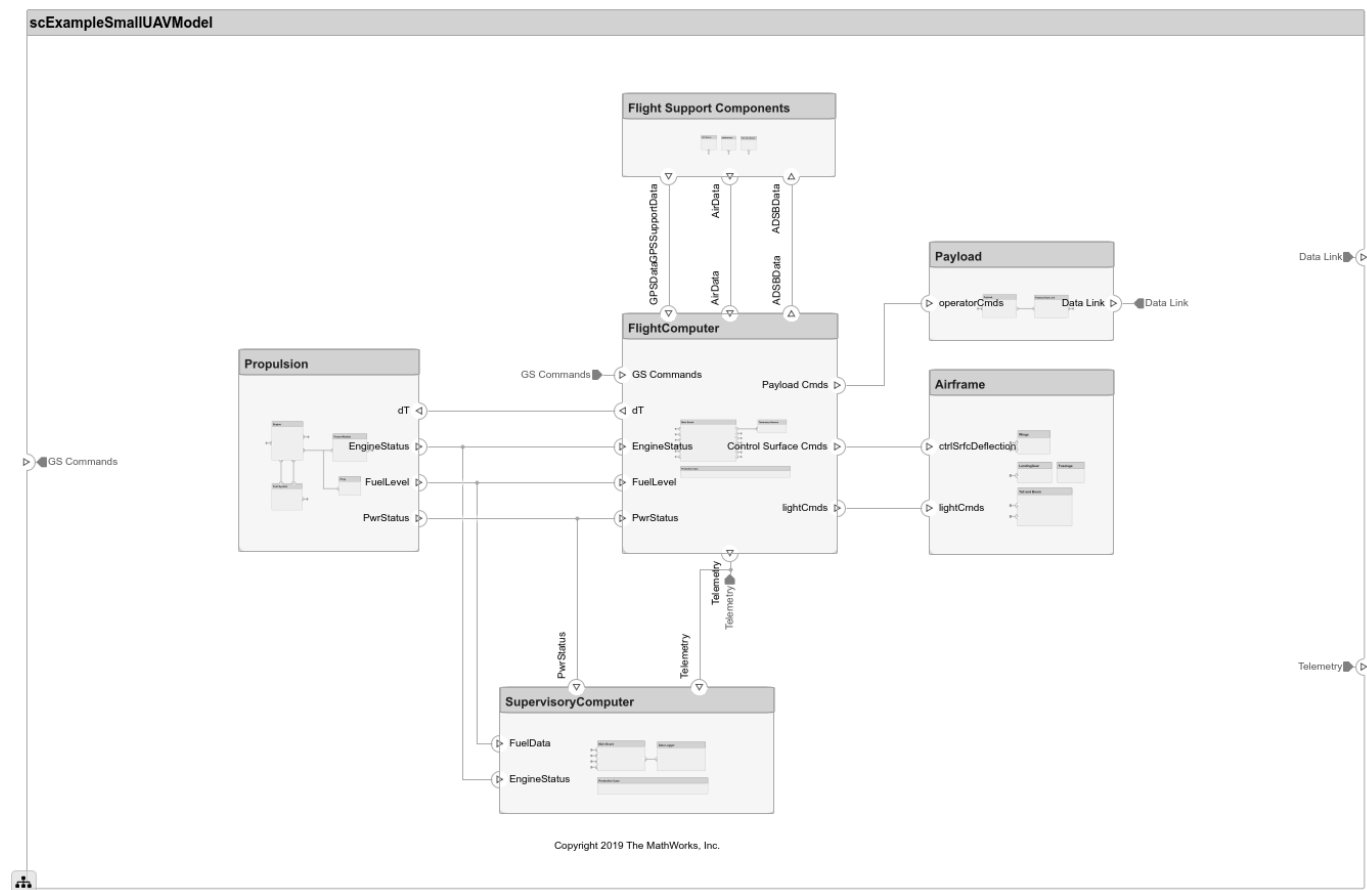
Overview

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
```

Starting: Simulink

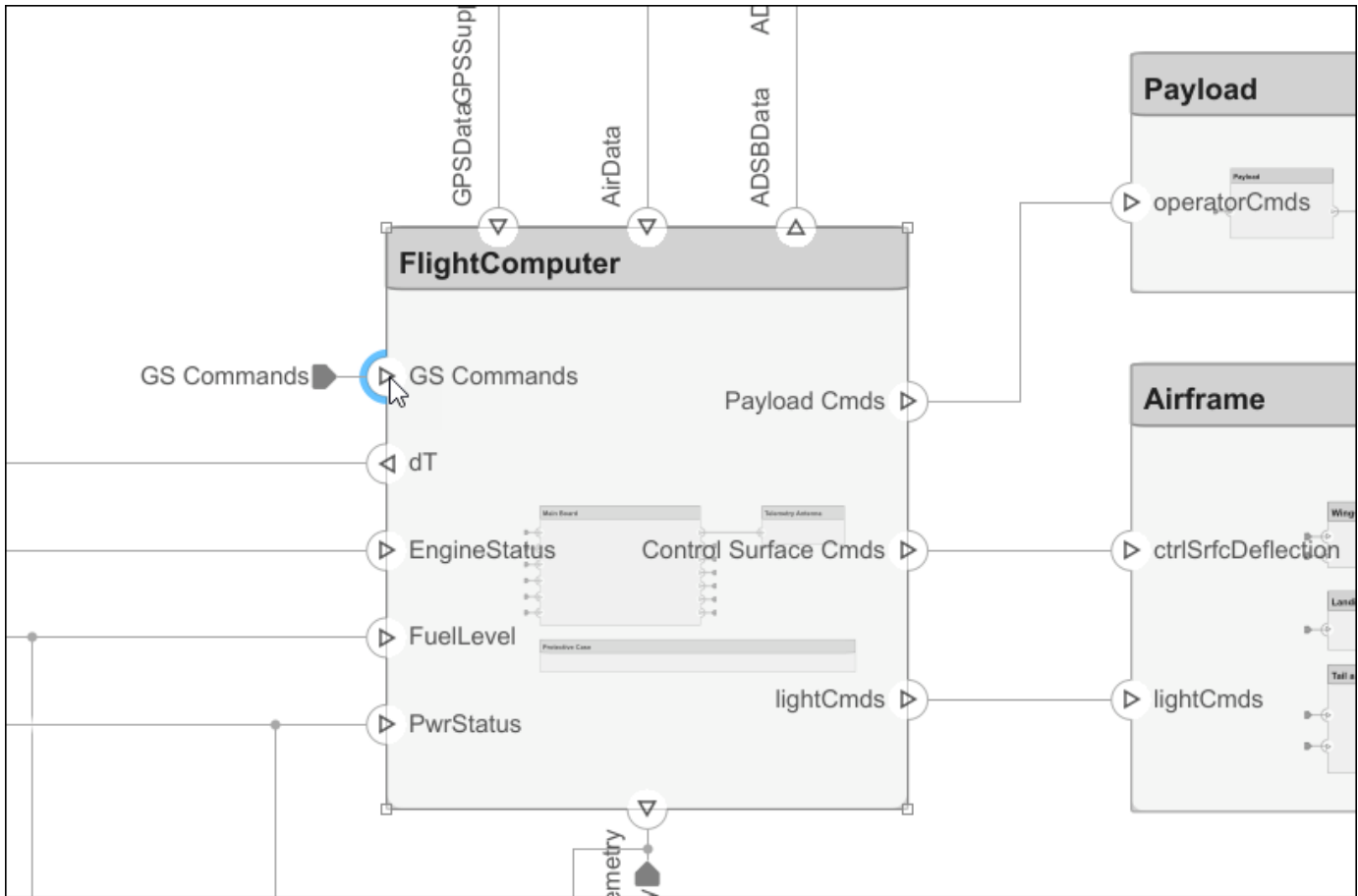


Each top-level component is decomposed into its subcomponents. Navigate through the hierarchy to view the composition for each component. The root component, `scExampleSmallUAVModel`, has input and output ports that represent data exchange between the system and its environment.

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. To open the Interface Editor, in the **Modeling** tab in the toolstrip, 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.

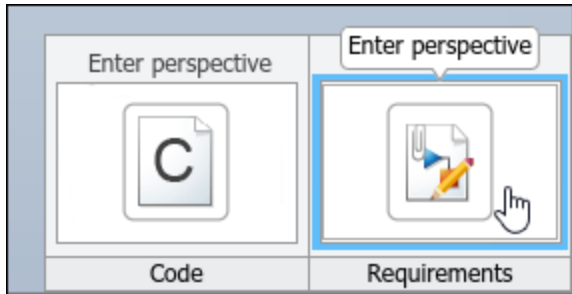


Interfaces							
	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
scExampleSmallUAVModel.sbx							
architecture_gsCommands							
apConfigParams (param_value_bus)	param_value_bus	1		real	0	0	
param_count	uint16	1		real	0	0	Total number of onboard parameters
param_id	int8	16		real	0	0	Onboard parameter id, terminated by NULL if the length is less than 16 human-
param_index	uint16	1		real	0	0	Index of this onboard parameter
param_type	uint8	1		real	0	0	Onboard parameter type: see the MAV_PARAM_TYPE enum for supported data
param_value	single	1		real	0	0	Onboard parameter value
gsCommands (gs_commands_bus)	gs_commands_bus	1		real	0	0	
RTB	uint8	1		real	0	0	Return to Base Command
U_c	single	1		real	0	0	Airspeed Commanded by the GS
guidanceMode	uint8	1		real	0	0	
h_c_midLevel	single	1		real	0	0	Commanded altitude when in Mid Level Commands Mode. Also used as the alti
isManualModeOn	uint8	1		real	0	0	
psiDot_c_midLevel	single	1		real	0	0	Turnrate command when in mid Level Commands guidance mode

Inspect Requirements

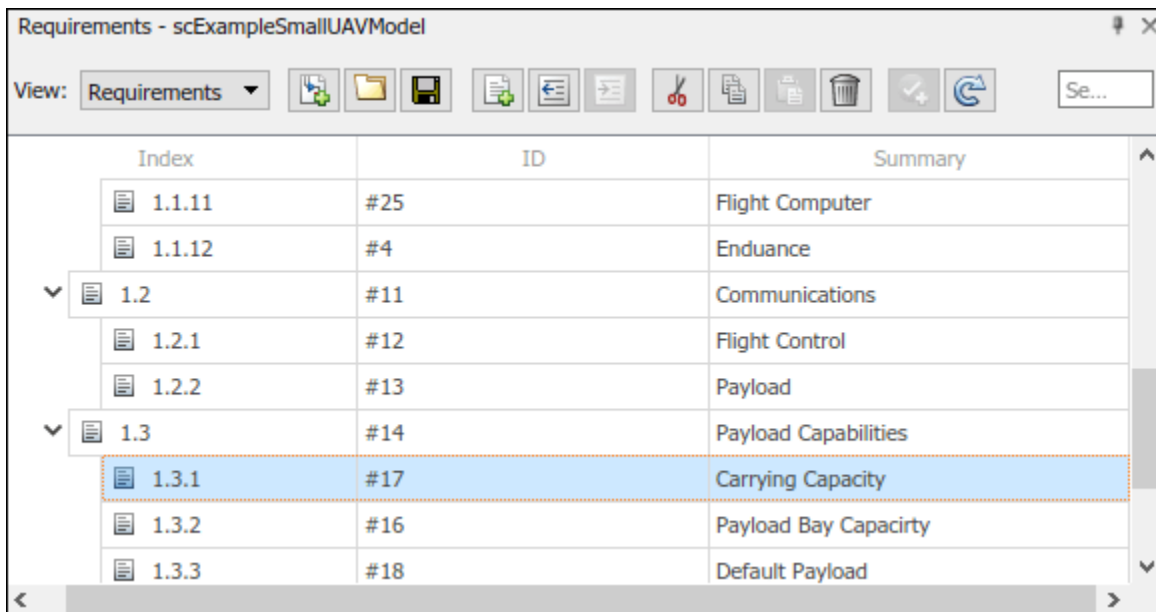
A Requirements Toolbox™ license is required to inspect requirements in a System Composer architecture model.

Components in the architecture model link to system requirements defined in `smallUAVReqs.s1reqx`. Open the **Requirements Manager**. In the bottom right corner of the model pane, click **Show Perspectives views**. 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.s1reqx`, select the Requirements Editor (Requirements Toolbox) 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.



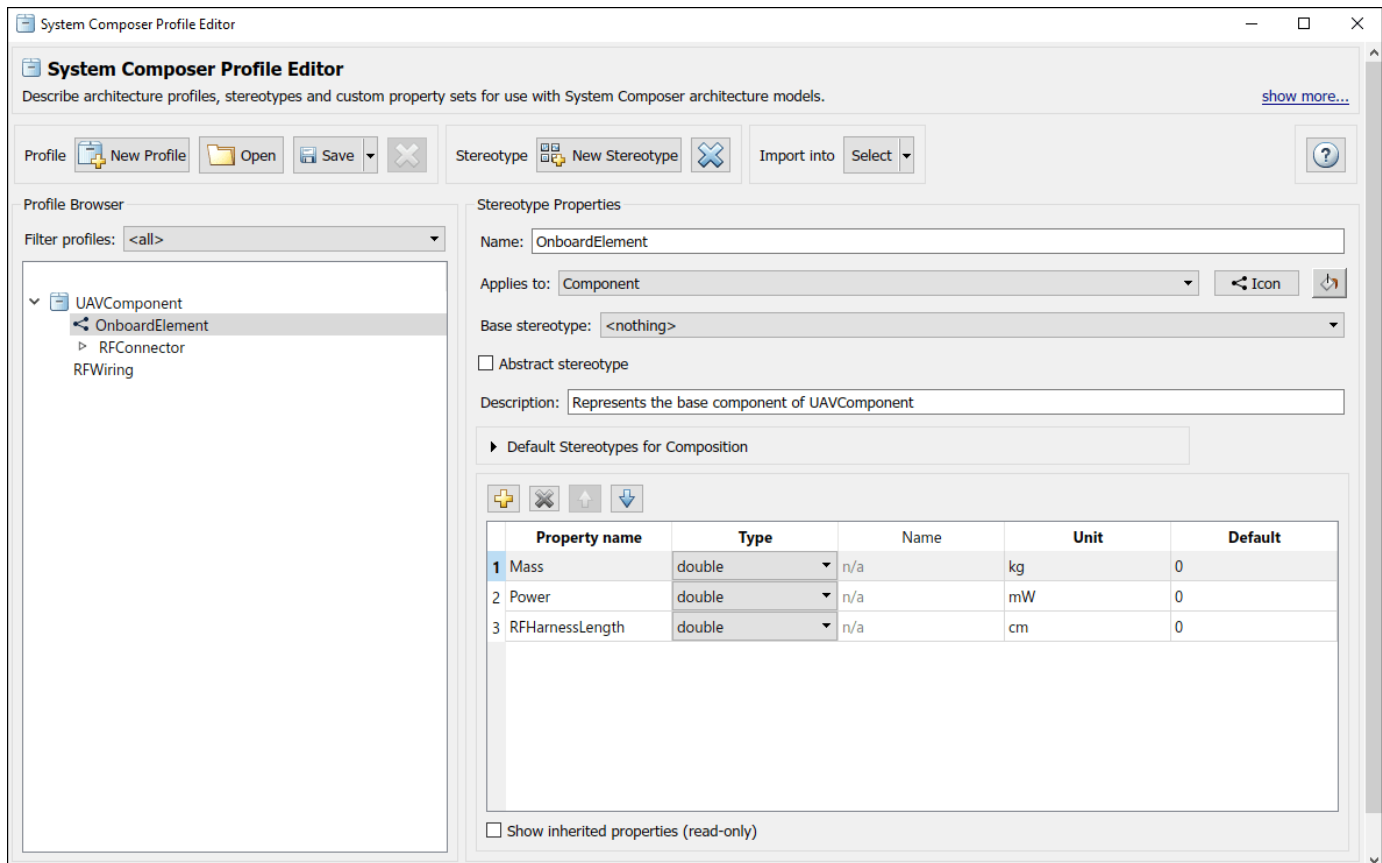
Define Profiles and Stereotypes

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 **Profile Editor**.

The display appears below.

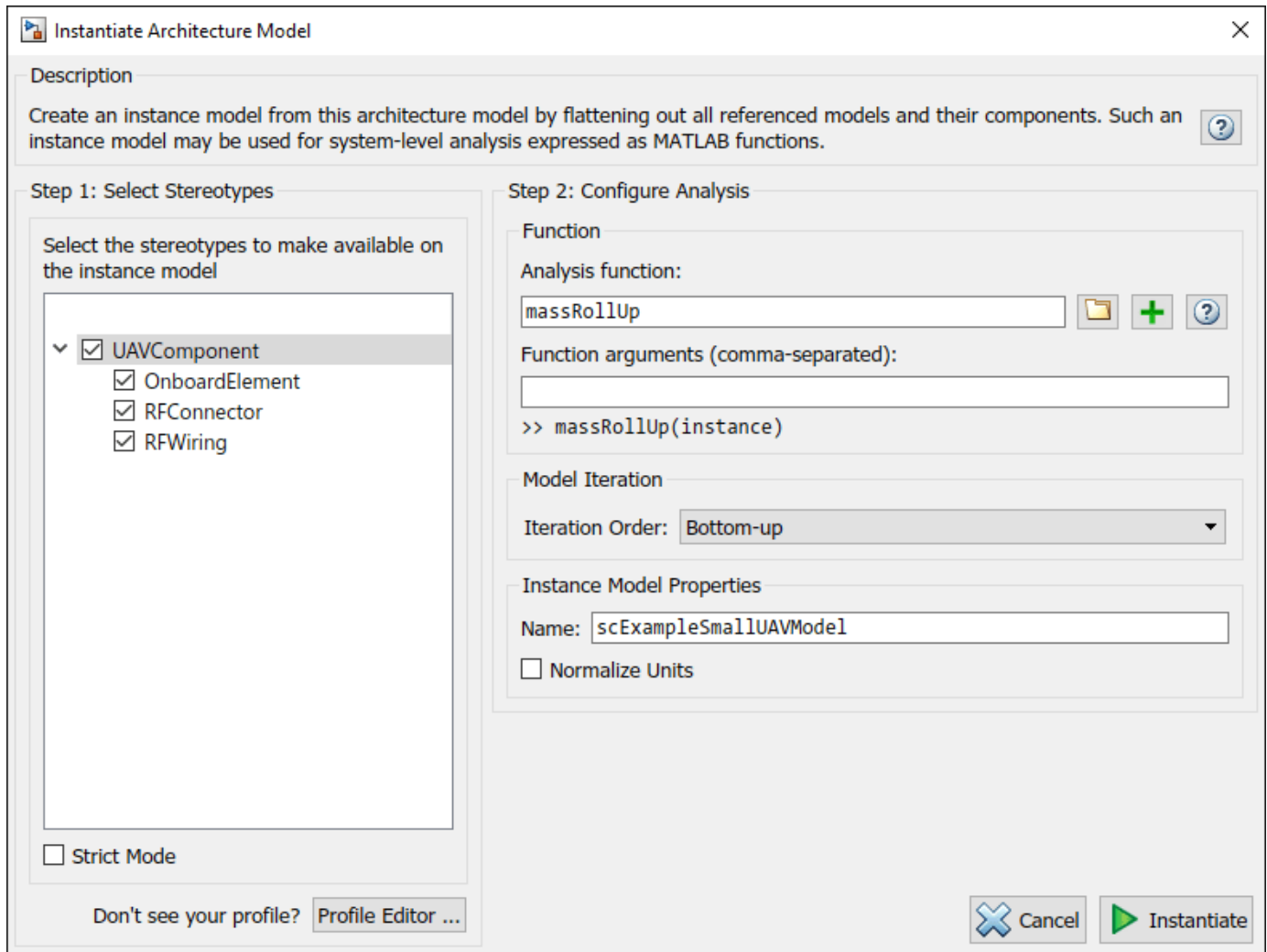


Analyze the Model

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.

To open the Instantiate Architecture Model tool, 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**.



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:GPSModuleData				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:GPSModuleData->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

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

More About

- “Create Interfaces” on page 3-5
- “Manage Requirements” on page 2-8
- “Define Profiles and Stereotypes” on page 4-2
- “Analyze Architecture” on page 7-2
- “Organize System Composer Files in Projects” on page 10-2

Requirements

- “Link and Trace Requirements” on page 2-2
- “Manage Requirements” on page 2-8
- “Update Reference Requirement Links from Imported File” on page 2-15

Link and Trace Requirements

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

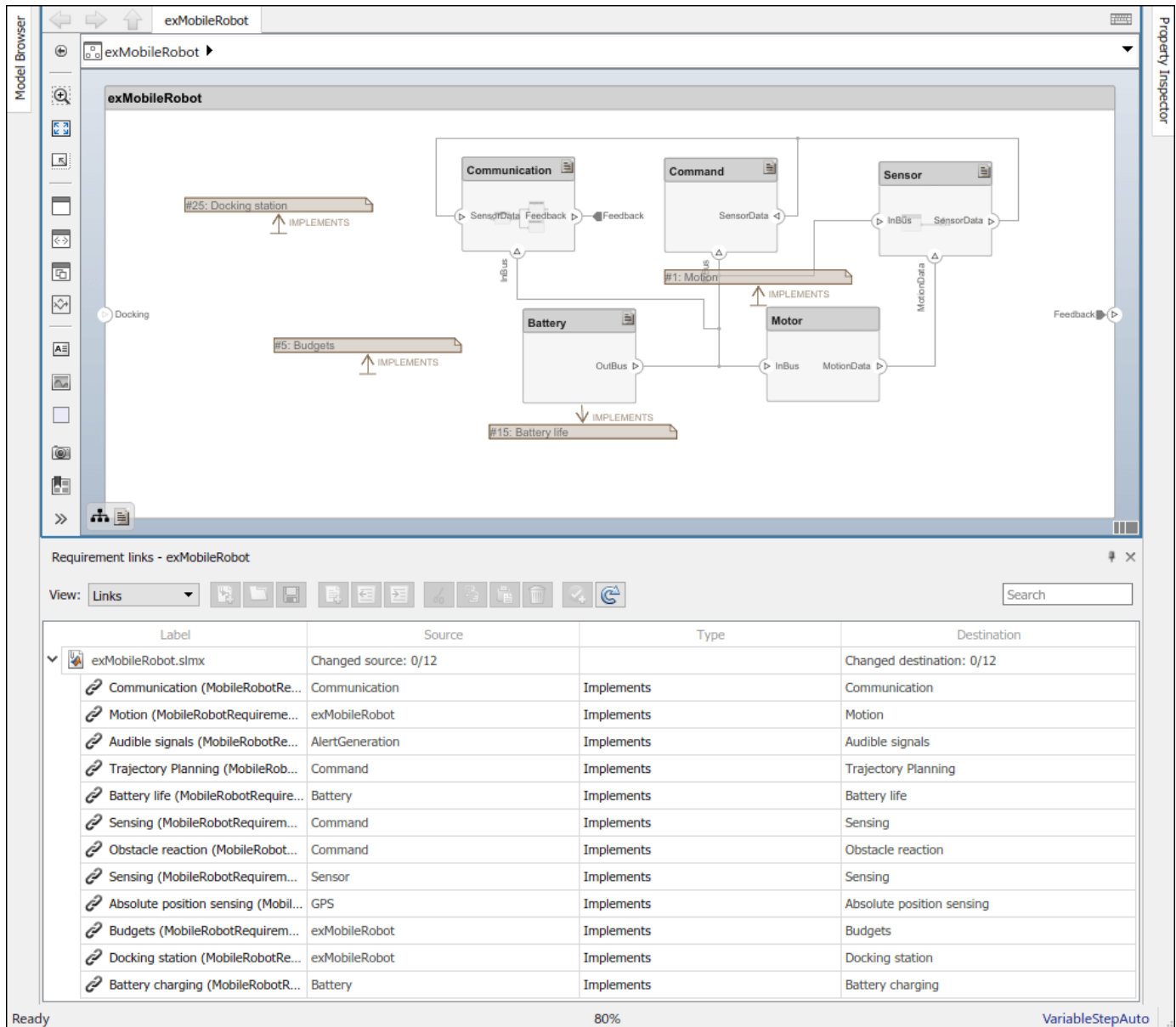
Allocate functional requirements to components to establish traceability. By creating a link between a component and the related requirement, you can track whether all requirements are represented in the architecture. You can also keep requirements and design in sync, for example, if a requirement changes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

A Requirements Toolbox license is required to link, trace, and manage requirements in System Composer™.

Open the model `exMobileRobot`.

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

Manage requirements and architecture together in the **Requirements Manager** from Requirements Toolbox. Navigate to **Apps > Requirements Manager**. You are now in the Requirements Perspective in System Composer.



Links can be created and managed through the Requirements Perspective. For more information, see “Manage Requirements” on page 2-8. This example shows an alternative approach using the Requirements Editor.

Open the requirements in the Requirements Editor (Requirements Toolbox).

```
slreq.open('MobileRobotRequirements');
```

Select the requirement to be linked.

The screenshot displays a requirements management interface. On the left is a tree view of requirements, and on the right is a detailed view for a selected requirement.

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

Details

▼ Properties

Type:

Index: 3.1

Custom ID:

Summary:

Description Rationale

B *I* U

Device shall convey operation errors via audible signals.

Keywords:

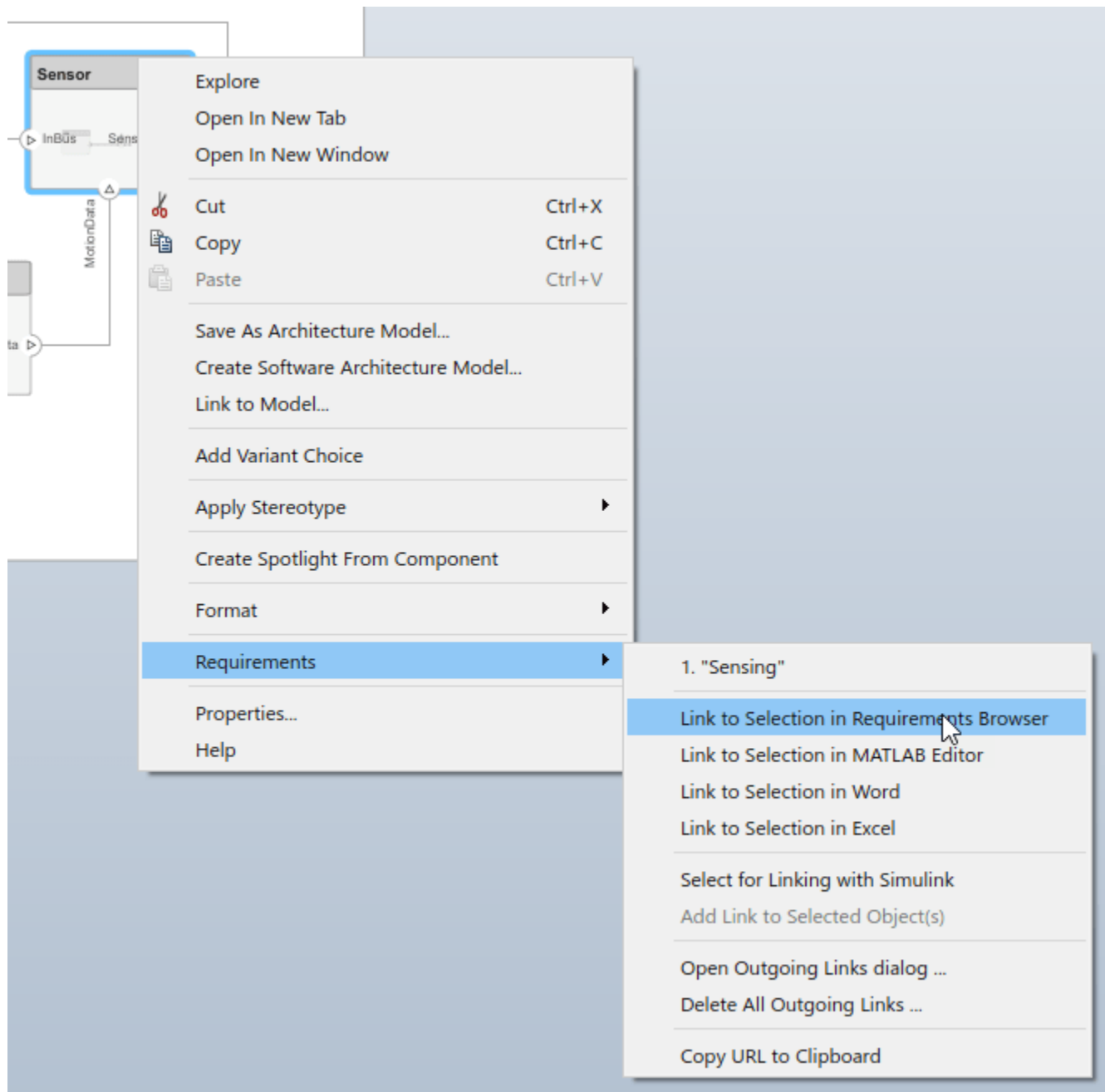
▶ Revision information:

▼ Links

← Implemented by:

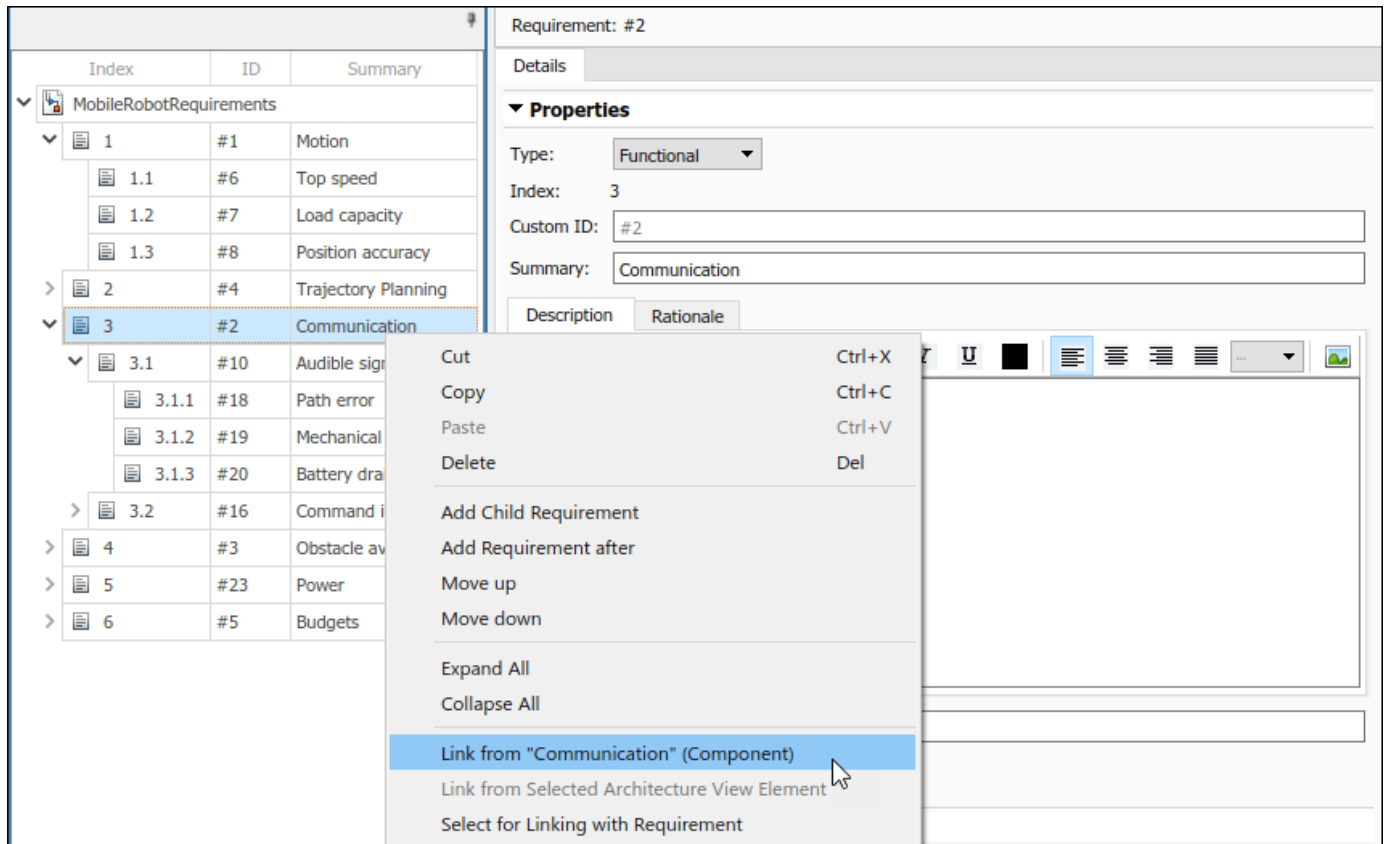
[AlertGeneration](#)

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




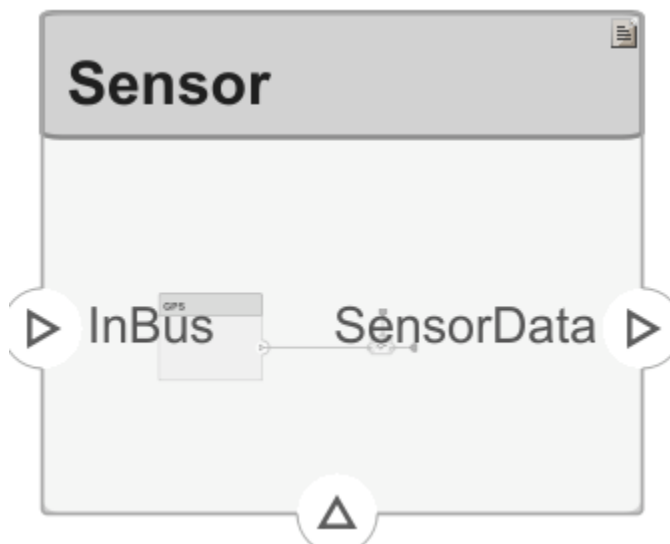
When you first link a requirement in an architecture model, a link set file with extension `.slmx` is created to store requirement links. The **Requirements** context menu displays the linked requirements.

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

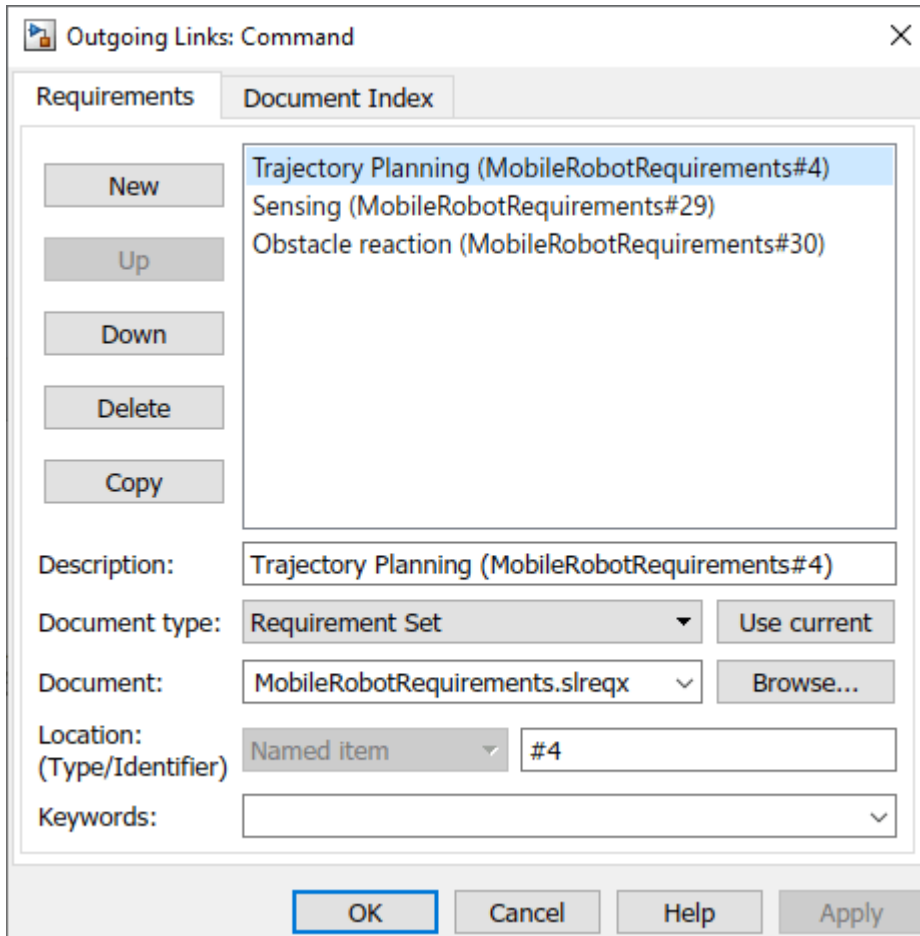


You can also create requirement links with blocks and subsystems in Simulink models. For more information, see “Link Blocks and Requirements” (Requirements Toolbox).

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 the Command component and select **Requirements > Open Outgoing Links dialog**. Here, you can create new requirements, delete existing ones, and change their order.



See Also

More About

- “Manage Requirements” on page 2-8
- “Organize System Composer Files in Projects” on page 10-2
- “View Requirements Toolbox Links Associated with Model Elements”
- “Design Insulin Infusion Pump Using Model-Based Systems Engineering” on page 7-23
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

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.

A Requirements Toolbox™ license is required to link, trace, and manage requirements in System Composer.

To enhance traceability of requirements, link system, functional, customer, performance, or design requirements to components and ports. Link requirements to each other to represent derived or allocated requirements. Manage requirements from the Requirements Manager on an architecture model or through custom views. Assign test cases to requirements using the **Test Manager** for verification and validation.

A Simulink Test™ license is required to use the Test Manager and to create test harnesses for components in System Composer.

A requirement set is a collection of requirements. You can structure the requirements hierarchically and link them to components or ports.

Use the **Requirements Editor** to edit and refine requirements in a requirement set. Requirement sets are stored in SLREQX files. You can create a new requirement set and author requirements using Requirements Toolbox, or import requirements from supported third-party tools.

A link is an object that relates two model-based design elements. A requirement link is a link where the destination is a requirement. You can link requirements to components or ports.

View links using the Requirements Perspective in System Composer. Select a requirement in the Requirements Browser to highlight the component or the port to which the requirement is assigned. Links are stored externally as SLMX files.

Mobile Robot Architecture Model

This example shows a mobile robot platform architecture.

Manage Requirements

Manage requirements and architecture together in the **Requirements Manager** from Requirements Toolbox. Navigate to **Apps > Requirements Manager**. You are now in the Requirements Perspective in System Composer.

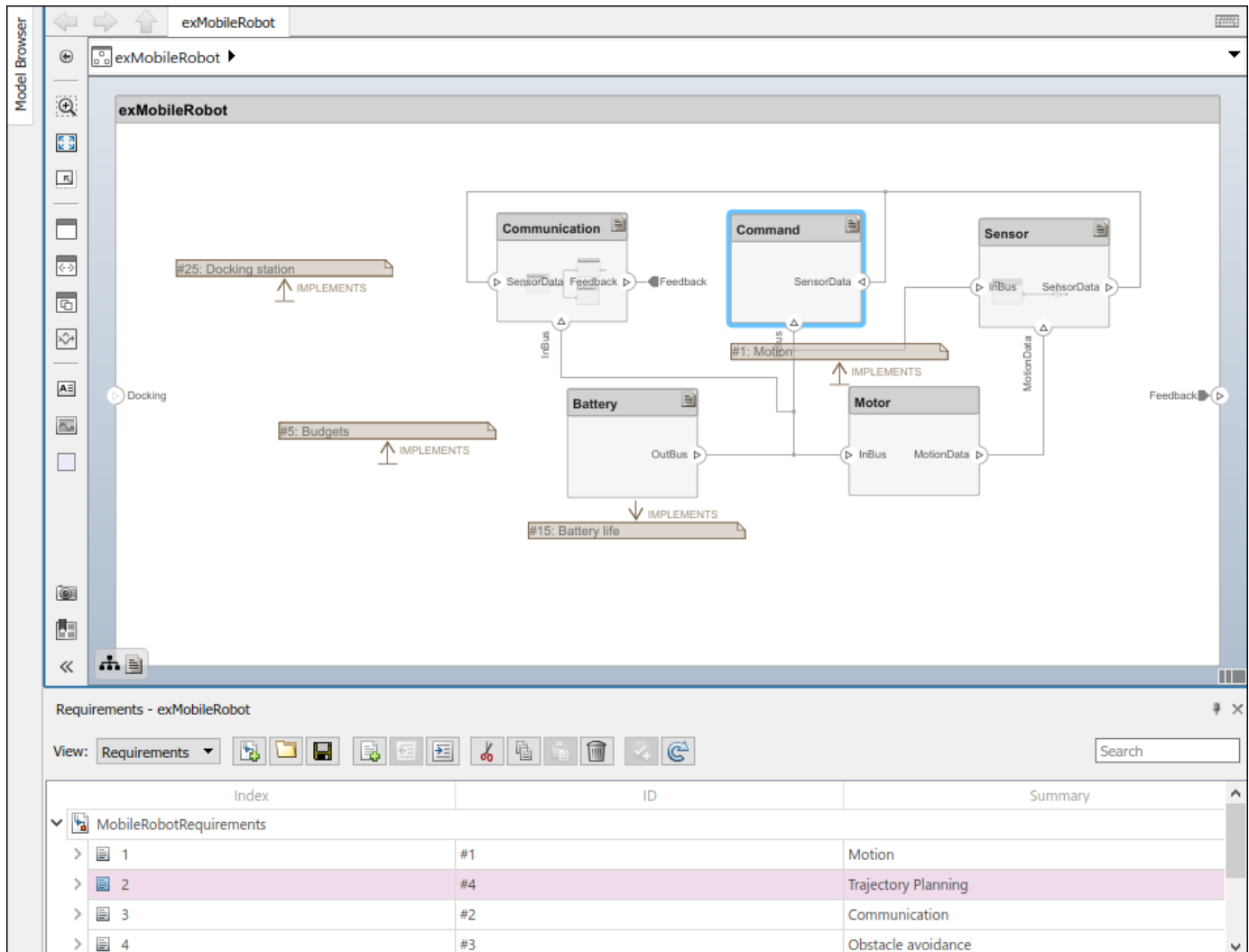
The screenshot displays the Requirements Perspective for the exMobileRobot system. The top portion shows a system architecture diagram with components: Communication, Command, Sensor, Battery, and Motor. The Communication component has ports for SensorData and Feedback. The Command component has a SensorData port. The Sensor component has ports for InBus and SensorData. The Battery component has an OutBus port. The Motor component has ports for InBus and MotionData. Connections are shown between these components, including InBus and MotionData links.

Below the diagram is a table titled "Requirements - exMobileRobot" showing a list of requirements. The table has columns for Index, ID, and Summary.

Index	ID	Summary
> [icon] MobileRobotRequirements		
> [icon] 1	#1	Motion
> [icon] 3	#2	Communication
> [icon] 4	#3	Obstacle avoidance
> [icon] 2	#4	Trajectory Planning
> [icon] 6	#5	Budgets
> [icon] 5	#23	Power

Trace Requirements

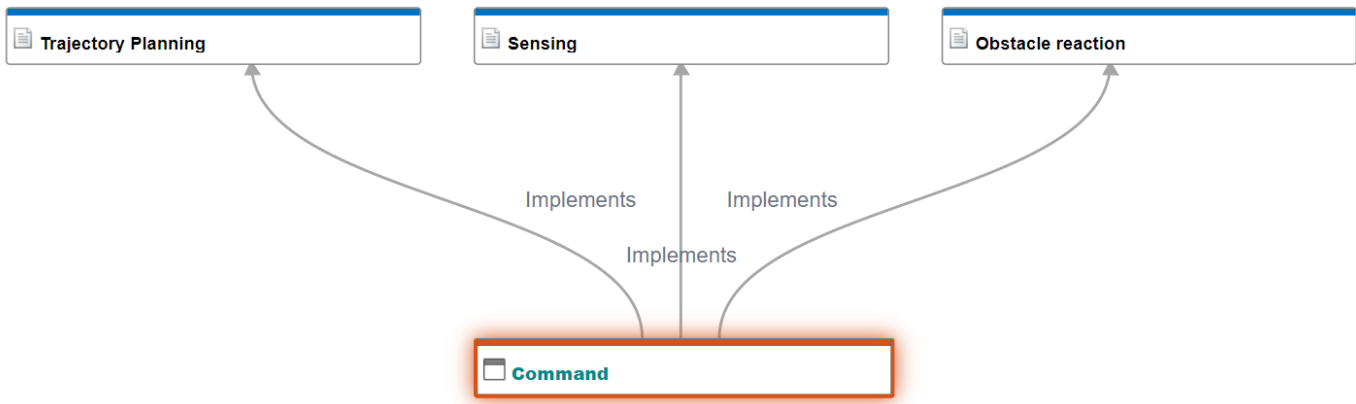
When you click a component in the Requirements Perspective, linked requirements are highlighted. Conversely, when you click a requirement, the linked components are shown.



Use Requirements Traceability Diagram

Visualize traceability of requirements and how they are related using a traceability diagram.

Change the **View** option on the Requirements Manager from Requirements to Links. Right-click the Trajectory Planning requirement link and select View Traceability Diagram.



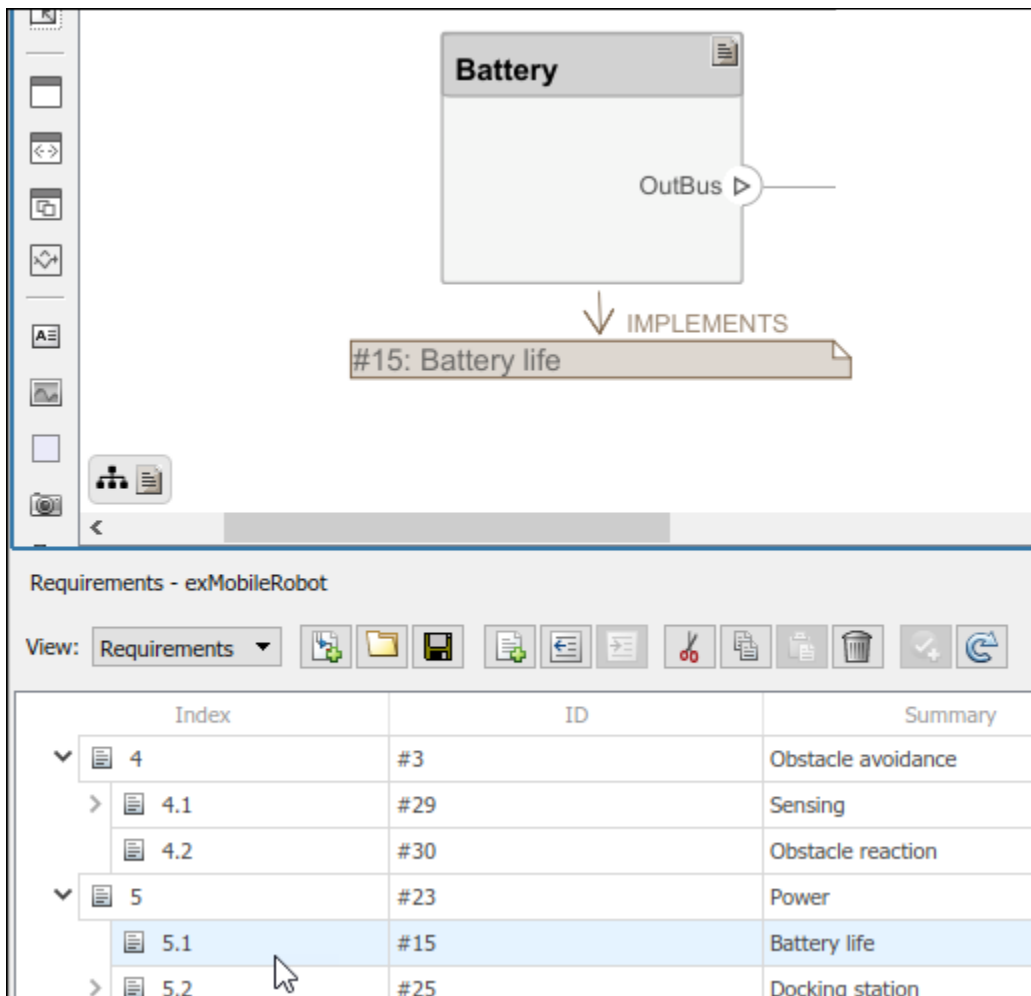
According to this traceability diagram, the **Command** component implements the three requirements **Trajectory Planning**, **Sensing**, and **Obstacle reaction**.

Change the **View** option on the Requirements Manager from **Links** back to **Requirements**.

For more information, see “Visualize Links with a Traceability Diagram” (Requirements Toolbox).

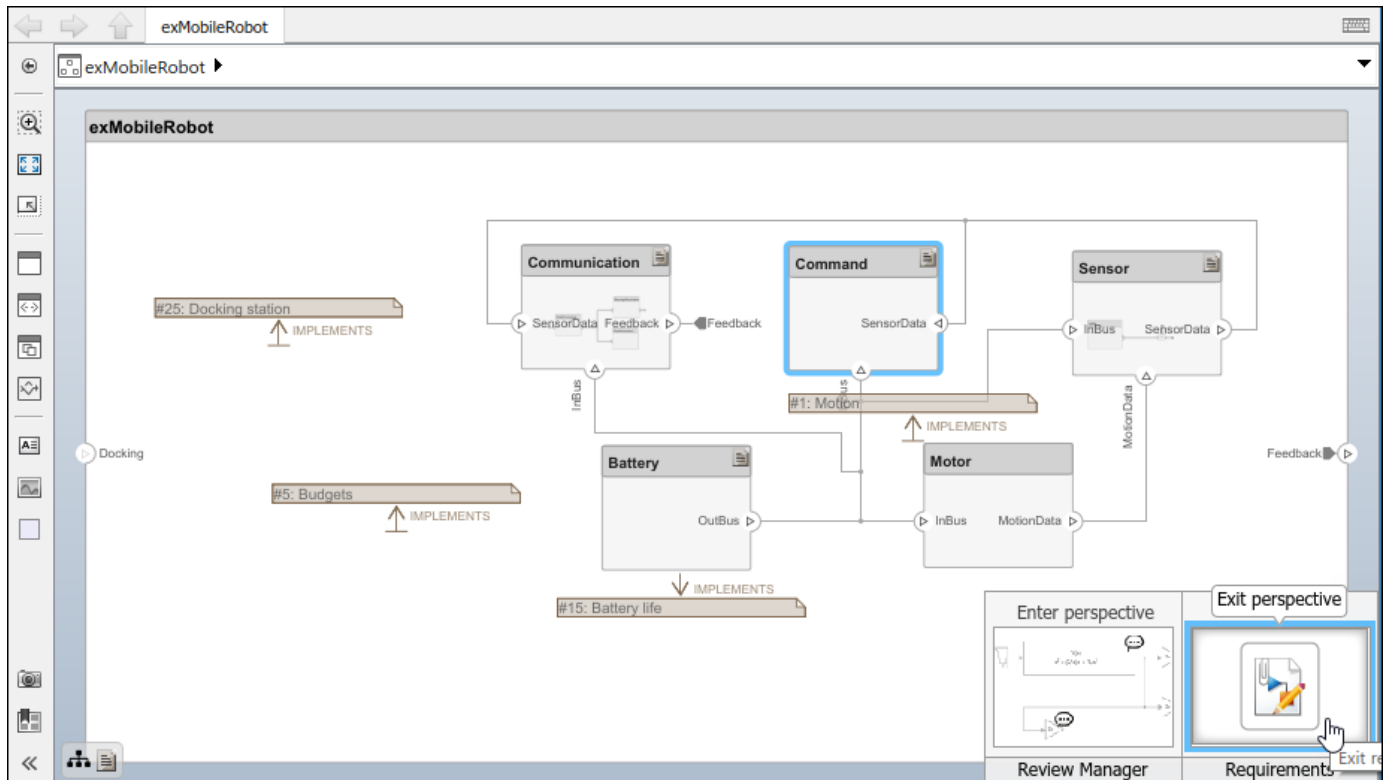
Link Requirements

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



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” (Requirements Toolbox). To integrate the requirement links to the model, see “Update Reference Requirement Links from Imported File”.

Verify and Validate Requirements Using Test Harnesses

A test harness is a model that isolates the component under test with inputs, outputs, and verification blocks configured for testing scenarios. You can create a test harness for a model component or for a full model. A test harness gives you a separate testing environment for a model or a model component.

For more information, see “Create a Test Harness” (Simulink Test).

Create a test harness for a System Composer component to validate simulation results and verify design. The **Interface Editor** is accessible in System Composer test harness models to enable behavior testing and implementation-independent interface testing.

Use Simulink Test to perform requirement-based testing workflows that include inputs, expected outputs, and acceptance criteria. For more information on using Simulink Test with Requirements Toolbox, see “Link to Test Cases from Requirements” (Requirements Toolbox).

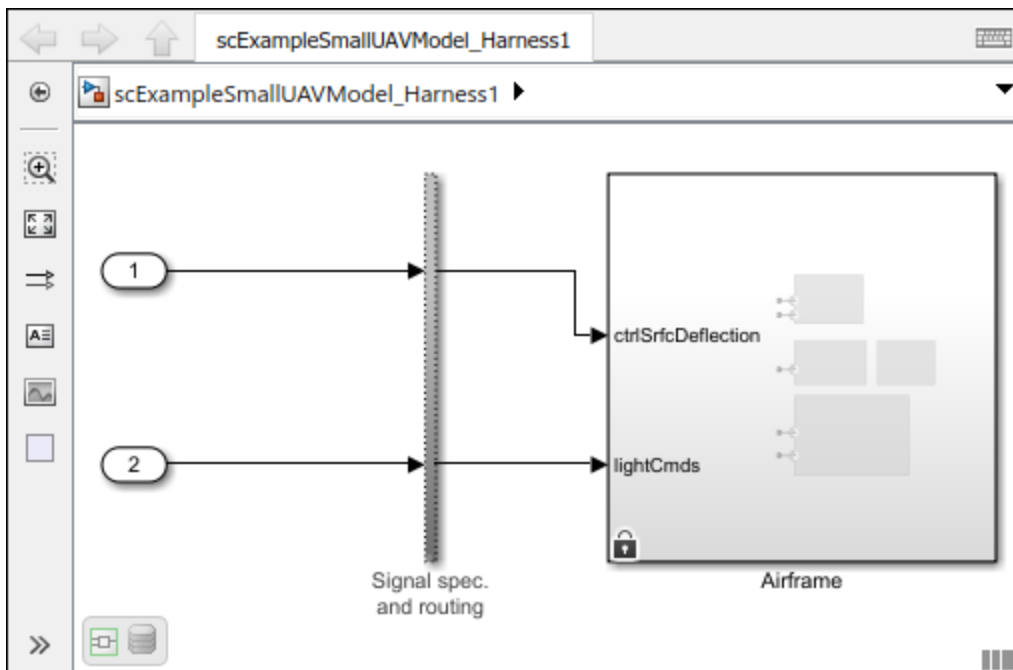
Note Test harnesses are not supported for Adapter blocks in architecture models or Component blocks that contain reference components in software architecture models.

This example uses the architecture model for an unmanned aerial vehicle (UAV) to create a test harness for a System Composer component. In the MATLAB Command Window, enter this command.

scExampleSmallUAV

To create a test harness for the **Airframe** component, right-click the component and select **Test Harness > Create for 'Airframe'**. In the Create Test Harness dialog box, specify the name of your test harness and click **OK**. Your test harness opens in a new window, and the **Harness** menu is available in the toolbar.

Tip If the model component is not fully wired and in an early step in the design process, you can select the **Advanced Properties** tab in the Create Test Harness dialog box and select **Create without compiling the model**.



Use the **Test Manager** with the test harness to create test files and test cases. For more information, see “Test Harness and Model Relationship” (Simulink Test) and “Create Test Harnesses and Select Properties” (Simulink Test).

See Also

More About

- “Link and Trace Requirements” on page 2-2
- “Import and Export Architectures” on page 11-2
- “Compose Architecture Visually” on page 1-2
- “Organize System Composer Files in Projects” on page 10-2
- “Design Insulin Infusion Pump Using Model-Based Systems Engineering” on page 7-23

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 Requirements Toolbox™ functionality.

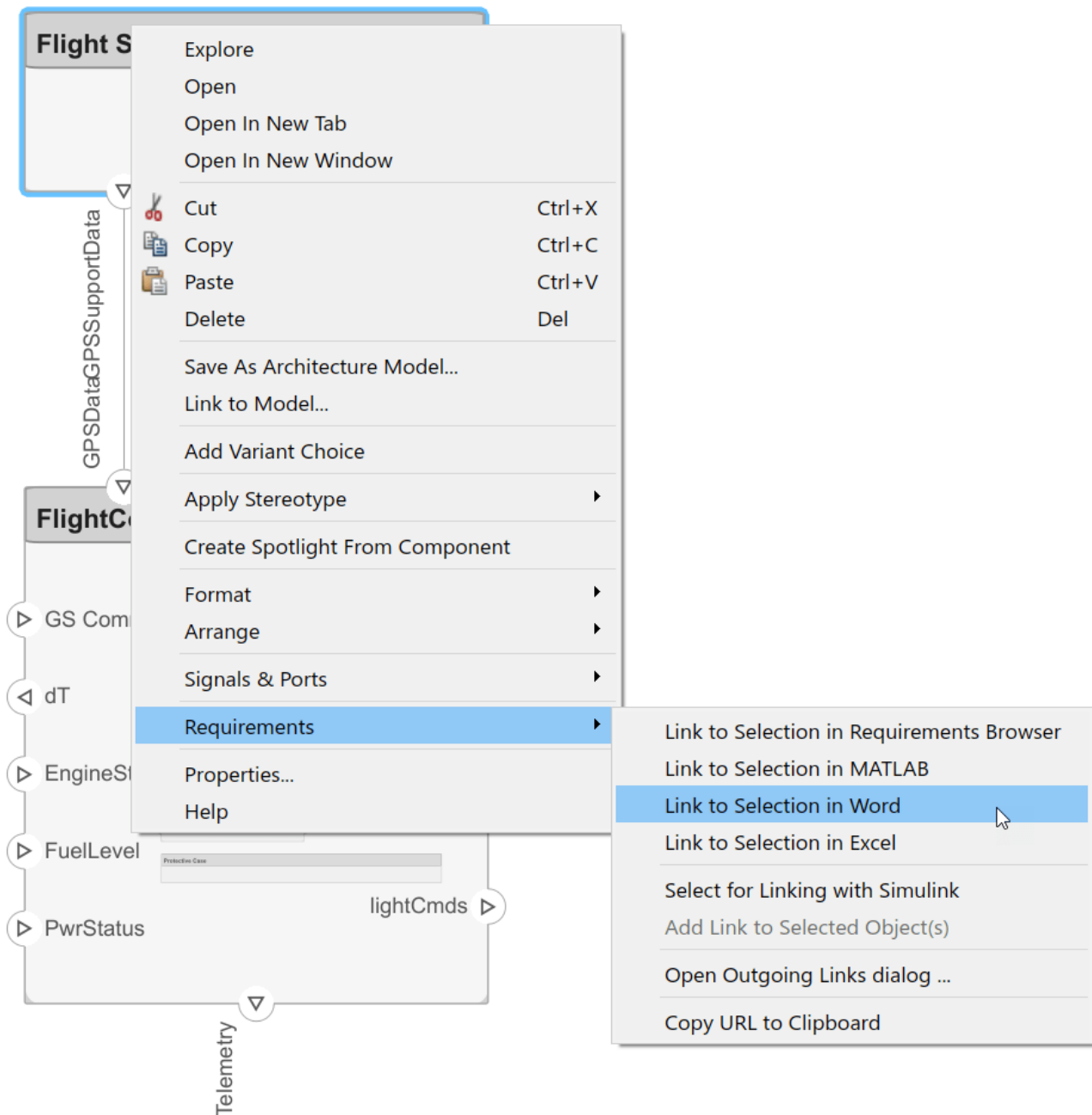
```
model = systemcomposer.openModel("reqImportExample");
```

Note: Importing or linking requirements may not work with a web-based Microsoft® Office file stored in SharePoint or OneDrive. Use a local copy of the file.

Import Requirement Links from Word File

Open the Microsoft® Word file `Functional_Requirements.docx` with the requirements listed. Highlight the requirement to link.

In the model, select the component to which to link the requirement. Right-click the component and select **Requirements > Link to Selection in Word**.



Export Model and Save to External File

Export the model and save to an external file.

```
exportedSet = systemcomposer.exportModel("reqImportExample");
SaveToExcel("exportedModel",exportedSet);
```

Import Requirement Links from File and Import to Model

Use the external file to import requirement links into another model.

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

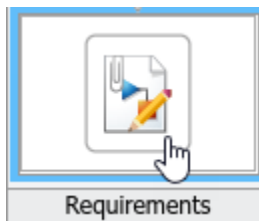
```
systemcomposer.importModel("reqNewExample", structModel.Components, ...
structModel.Ports, structModel.Connections, structModel.Interfaces, structModel.RequirementLinks);
```

Update Links to Reference Requirements

To integrate the requirement links to the model, update references within the model.

```
systemcomposer.updateLinksToReferenceRequirements("reqNewExample", "linktype_rmi_word", "Functiona
```

Open the **Requirements** perspective from the bottom right corner of the model palette to view the requirements.



See Also

[importModel](#) | [exportModel](#) | [updateLinksToReferenceRequirements](#)

More About

- “Link and Trace Requirements” on page 2-2
- “Manage Requirements” on page 2-8
- “Import and Export Architecture Models” on page 11-5
- “Custom Link Types” (Requirements Toolbox)

Interface Management

- “Define Port Interfaces Between Components” on page 3-2
- “Create Interfaces” on page 3-5
- “Assign Interfaces to Ports” on page 3-10
- “Interface Adapter” on page 3-16
- “Manage Interfaces with Data Dictionaries” on page 3-20
- “Reference Data Dictionaries” on page 3-23

Define Port Interfaces Between Components

A system engineering solution in System Composer 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.

A data interface defines the kind of information that flows through a port. The same interface can be assigned to multiple ports. A data interface can be composite, meaning that it can include data elements that describe the properties of an interface signal.

Data interfaces represent the information that is shared through a connector and enters or exits a component through a port. Use the **Interface Editor** to create and manage data interfaces and data elements and store them in an interface data dictionary for reuse between models.

A data element describes a portion of an interface, such as a communication message, a calculated or measured parameter, or other decomposition of that interface.

Data interfaces are decomposed into data elements:

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

A value type can be used as a port interface to define the atomic piece of data that flows through that port and has a top-level type, dimension, unit, complexity, minimum, maximum, and description.

You can also assign the type of data elements in data interfaces to value types. Add value types to data dictionaries using the **Interface Editor** so that you can reuse the value types as interfaces or data elements.

Use interfaces to describe information transmitted across connections through ports between components.

- “Create Interfaces” on page 3-5: Design interfaces and nested interfaces in the **Interface Editor** with data interfaces, data elements, and value types.
- “Assign Interfaces to Ports” on page 3-10: Assign data interfaces and data elements to ports. Define owned interfaces local to ports.
- “Manage Interfaces with Data Dictionaries” on page 3-20: Save external interface data dictionaries to reuse between different models, link data dictionaries to architecture models, and delete data interfaces from data dictionaries.
- “Reference Data Dictionaries” on page 3-23: Reference data dictionaries so you can selectively share interface definitions among models. Manage referenced data dictionaries in the Model Explorer.
- “Interface Adapter” on page 3-16: Use an Adapter block to help connect two components with incompatible port interfaces by mapping between the two interfaces. Use the Interface Adapter dialog by double-clicking the Adapter block to map between interfaces, apply an interface conversion that breaks algebraic loops with unit delays, or insert a rate transition for different sample time rates. When output interfaces are undefined, you can use input interfaces in bus creation mode of the Interface Adapter to author owned output interfaces as you work.

The architecture model below represents an adapter, an interface data dictionary, a data interface, a data element, and a value type.

The diagram shows a block diagram with the following components and connections:

- PowerSource** block (left) connected to a **PowerSource** block (middle).
- The middle **PowerSource** block is connected to a **Wire** block (middle).
- The **Wire** block is connected to an **adapter** block (middle-right), which has **In** and **Out** ports.
- The **adapter** block is connected to another **Wire** block (right).
- The **Wire** block is connected to a **Robot** block (right).

The **Interfaces** table below provides details on the interface definitions:

	Type	Units
archDictionary.sidd (interface data dictionary)		
Charger (data interface)		
Voltage (VoltageType) (data element)	VoltageType	W
Wiring (data element)	double	m
VoltageType (value type)	double	W
RobotPower		
RobotVoltage (VoltageType)	VoltageType	W
Wires	double	m

Note System Composer interfaces mirror Simulink interfaces that use buses and value types. For more information, see “Simplify Subsystem and Model Interfaces with Buses”, “Specify Application-Specific Signal Properties”, and “Describe Component Behavior Using Simulink” on page 5-2.

See Also

More About

- “Specify Physical Interfaces on Ports” on page 5-76

- “Author Service Interfaces for Client-Server Communication” on page 8-41
- “Modeling System Architecture of Small UAV” on page 1-32
- “Modeling System Architecture of Keyless Entry System” on page 9-25

Create Interfaces

In this section...
“Mobile Robot Architecture Model” on page 3-5
“Open Interface Editor” on page 3-5
“Create Composite Data Interfaces” on page 3-6
“Create Value Types as Interfaces” on page 3-7
“Nest Interfaces to Reuse Data” on page 3-8

You can create interfaces between components in System Composer to structure transmitted data. Use composite data interfaces with data elements or value types to manage data defined on ports. Assign a data interface or value type to a data element so the data element inherits attributes and reuses data. Use the model below as a starting point before adding interfaces using the **Interface Editor**. For interfaces terminology, see “Define Port Interfaces Between Components” on page 3-2.

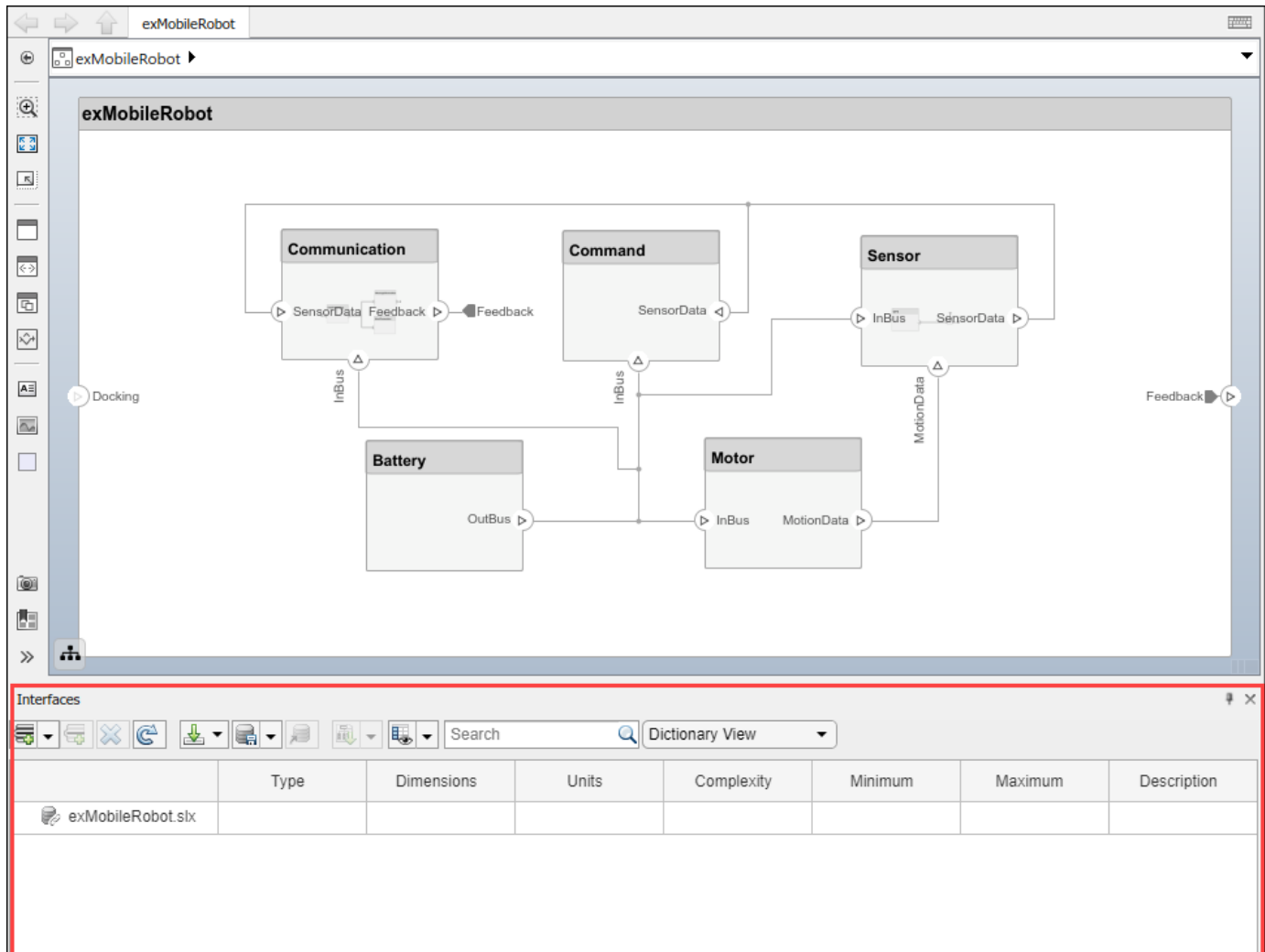
To manage interfaces shared between models in data dictionaries, see “Manage Interfaces with Data Dictionaries” on page 3-20. For information on physical interfaces, see “Specify Physical Interfaces on Ports” on page 5-76.

Mobile Robot Architecture Model

This example shows a mobile robot platform architecture.

Open Interface Editor

To open the **Interface Editor**, navigate to **Modeling > Interface Editor**. The **Interface Editor** will open at the bottom of the canvas.



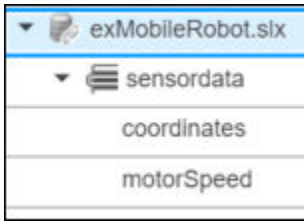
Note The System Composer **Interface Editor** is a web-based widget and might appear blank when you first launch it. If this occurs, save the model and relaunch MATLAB with the command line option `-cefdisablegpu`.


Create Composite Data Interfaces

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

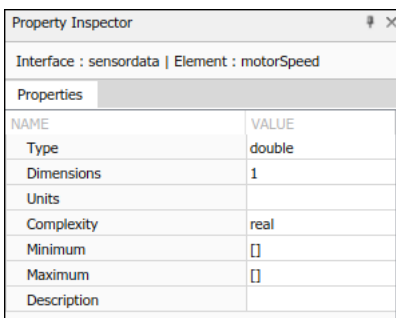


To add a data element to the data interface, click the  icon. Data interface and data element names must be valid MATLAB variable names.



You can delete data interfaces and data 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 data element and select **Inspect Properties**. For data interfaces, use the Property Inspector to apply stereotypes.

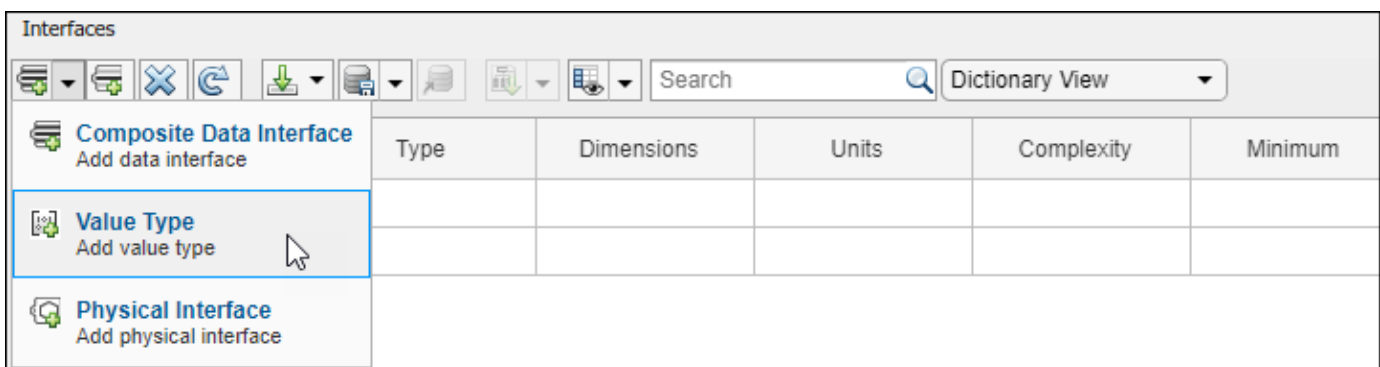


For a comparative view, you can edit data element properties from the relevant Interface Editor columns.

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

Create Value Types as Interfaces

To add a value type in the **Interface Editor**, select the down arrow next to the  icon and select **Value Type**. Name the value type motorSpeedType. Value type names must be valid MATLAB variable names.



Right-click the `motorSpeed` data element and select **Set 'Type' > motorSpeedType**. The data element `motorSpeed` is assigned to the value type `motorSpeedType`.

	Type	Dimensions	Units	Complexity
exMobileRobot.slx				
sensordata				
coordinates	double	1		real
motorSpeed	double	1	m/s	real
motorSpeedType	double	1		real

Any data changes on the `motorSpeedType` value type is propagated to the `motorSpeed` data element. You can reuse value types any number of times. Data changes on a value type will propagate to each data element that uses the value type.

Nest Interfaces to Reuse Data


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

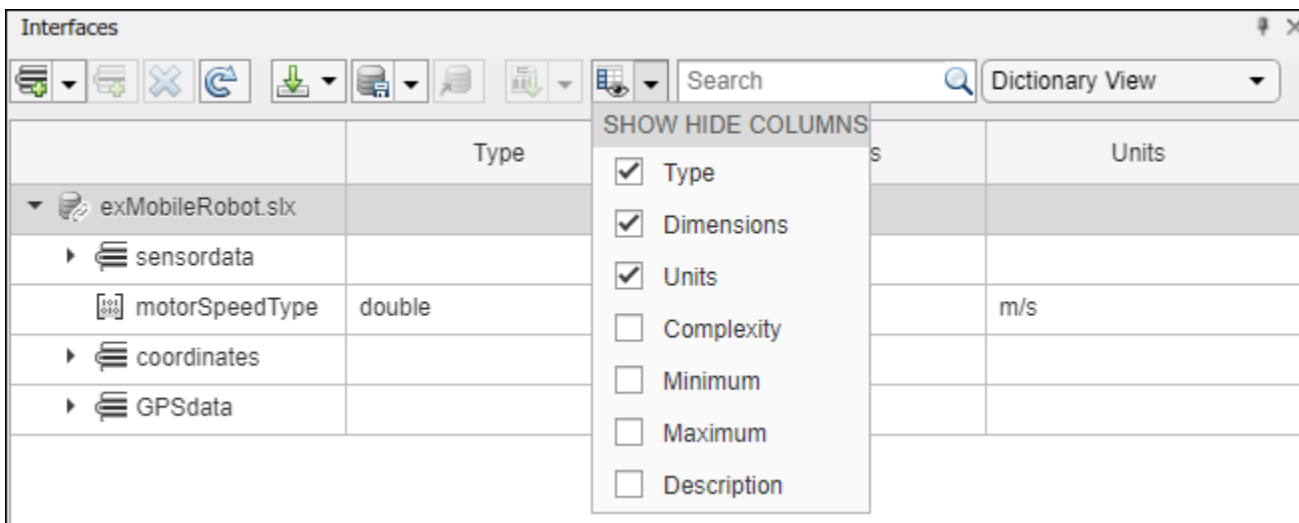
For example, let `coordinates` be a data interface that consists of `x`, `y`, and `z` coordinates. The `GPSdata` data interface includes `location` and a `timestamp`. If the `location` data element is in the same format as the `coordinates` interface, you can set its type to `coordinates`. Right-click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

	Type	Dimensions	Units	Complexity	Minimum	Maximum
exMobileRobot.slx						
sensordata						
coordinates	double	3	cm	real	0	100
motorSpeed (motorSpeedType)	motorSpeedType	1	m/s	real	0	35
motorSpeedType	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	[]	[]
location	double	1		real	[]	[]

The nested data interface displays the inherited data elements.

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

Note To change the number of columns that display in the **Interface Editor**, click the  icon. Select or clear the desired columns to show or hide them.



See Also

Functions

addInterface | removeInterface | addElement | removeElement | connect | setInterface | addValueType

Blocks

Component

More About

- “Assign Interfaces to Ports” on page 3-10
- “Interface Adapter” on page 3-16
- “Manage Interfaces with Data Dictionaries” on page 3-20
- “Specify Physical Interfaces on Ports” on page 5-76
- “Modeling System Architecture of Small UAV” on page 1-32

Assign Interfaces to Ports

In this section...

“Mobile Robot Architecture Model with Interfaces” on page 3-10

“Associate Ports with Interfaces in Property Inspector” on page 3-10

“Define Owned Interfaces Local to Ports” on page 3-11

“Select Multiple Ports and Assign Data Interface” on page 3-13

“Specify Source Element or Destination Element for Ports” on page 3-14

A port interface describes the data that can be passed between ports in a System Composer architecture model. Data elements within the interface describe characteristics of the data transmitted across the interface. Data elements can describe the composition of a data interface, messages transmitted, or data structures shared between components. For interfaces terminology, see “Define Port Interfaces Between Components” on page 3-2.

This topic will show you how to:

- Use the Property Inspector to assign data interfaces to one port at a time or the **Interface Editor** to assign data interfaces to multiple ports.
- Manage owned interfaces that are local to a port and not shared in a data dictionary.
- Assign interfaces to multiple ports at the same time.
- Connect components through ports and specify the source element or the destination element for the connection.

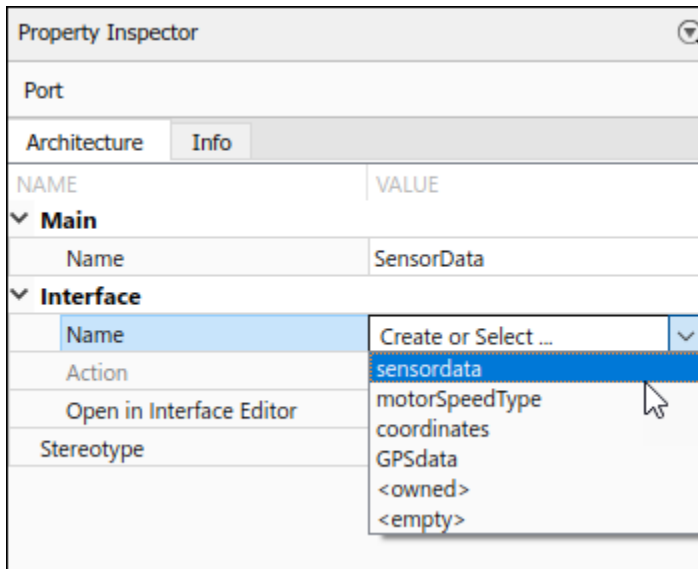
Incompatible data interfaces on either end of a connection can be reconciled with an Adapter block using the “Interface Adapter” on page 3-16. To manage interfaces shared between models in data dictionaries, see “Manage Interfaces with Data Dictionaries” on page 3-20. For information on physical interfaces, see “Specify Physical Interfaces on Ports” on page 5-76.

Mobile Robot Architecture Model with Interfaces

This example shows a mobile robot hardware architecture with interfaces defined.

Associate Ports with Interfaces in Property Inspector

To assign data interfaces or value types to one port at a time, use the Property Inspector. To open the Property Inspector, navigate to **Modeling > Design > Property Inspector**. To show the SensorData port properties, select the port in the model. Expand **Interface**, and from the **Name** list, select `sensordata` to associate the `sensordata` interface with the SensorData port.



Define Owned Interfaces Local to Ports

You can select a value type or data interface from the model data dictionary in the Property Inspector, or you can create an owned interface.

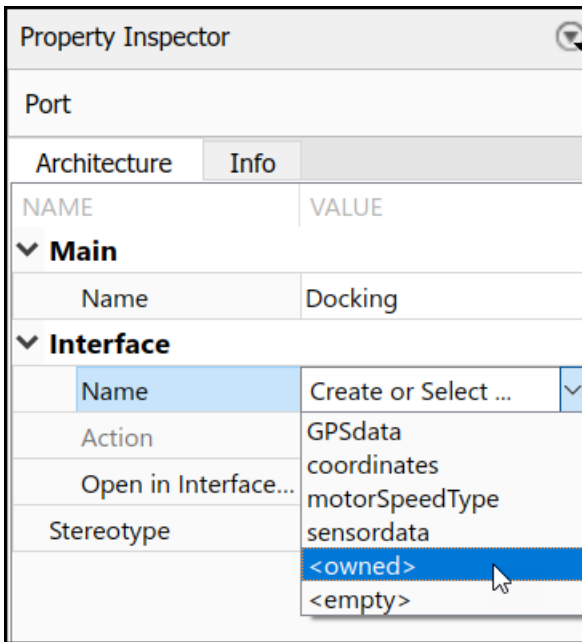
An owned interface is an interface that is local to a specific port and not shared in a data dictionary or the model dictionary.

Create an owned interface to represent a value type or data interface that is local to a port.

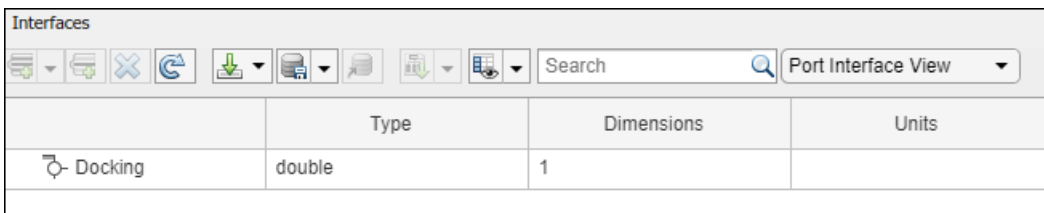
Note Owned interfaces and value types do not have their own names because they are local to a port and not shared. The name of the owned interface is derived from the port name.

Manage Owned Interfaces Using Property Inspector

You can edit the data for the owned interface in the Property Inspector. Select the **Docking** architecture port. In the Property Inspector, under **Interface**, from the **Name** list, select **<owned>**.



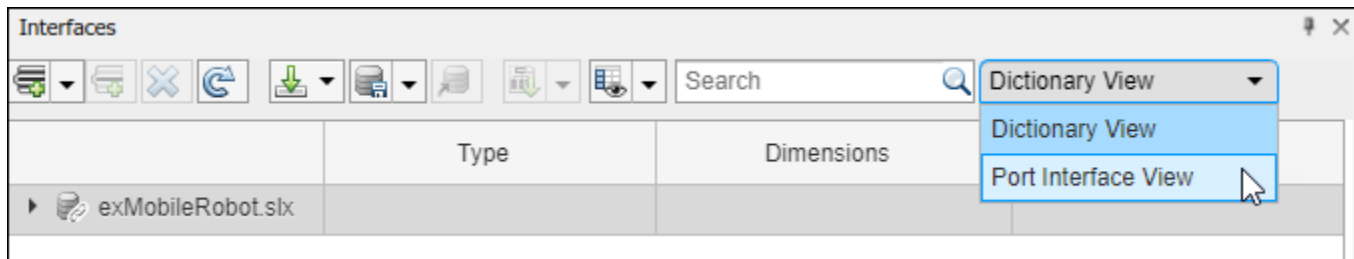
By default, the owned interface `Docking` becomes an owned value type. Edit interface attributes directly in the Property Inspector, or select `Open in Interface Editor` to edit the owned value type interface.



To convert the owned value type into an owned data interface, click to add a data element.

Manage Owned Interfaces Using Interface Editor

You can also work exclusively from the **Interface Editor**. Select the component port named `Feedback`. In the **Interface Editor**, change from `Dictionary View` to `Port Interface View`.



Click to add data elements to the owned data interface.

	Type	Dimensions	Units
Feedback			
ComputedSignal	double	1	
Userinput	double	1	

To convert the owned data interface to an owned value type, change the **Type** for Feedback to a valid MATLAB data type, such as double.

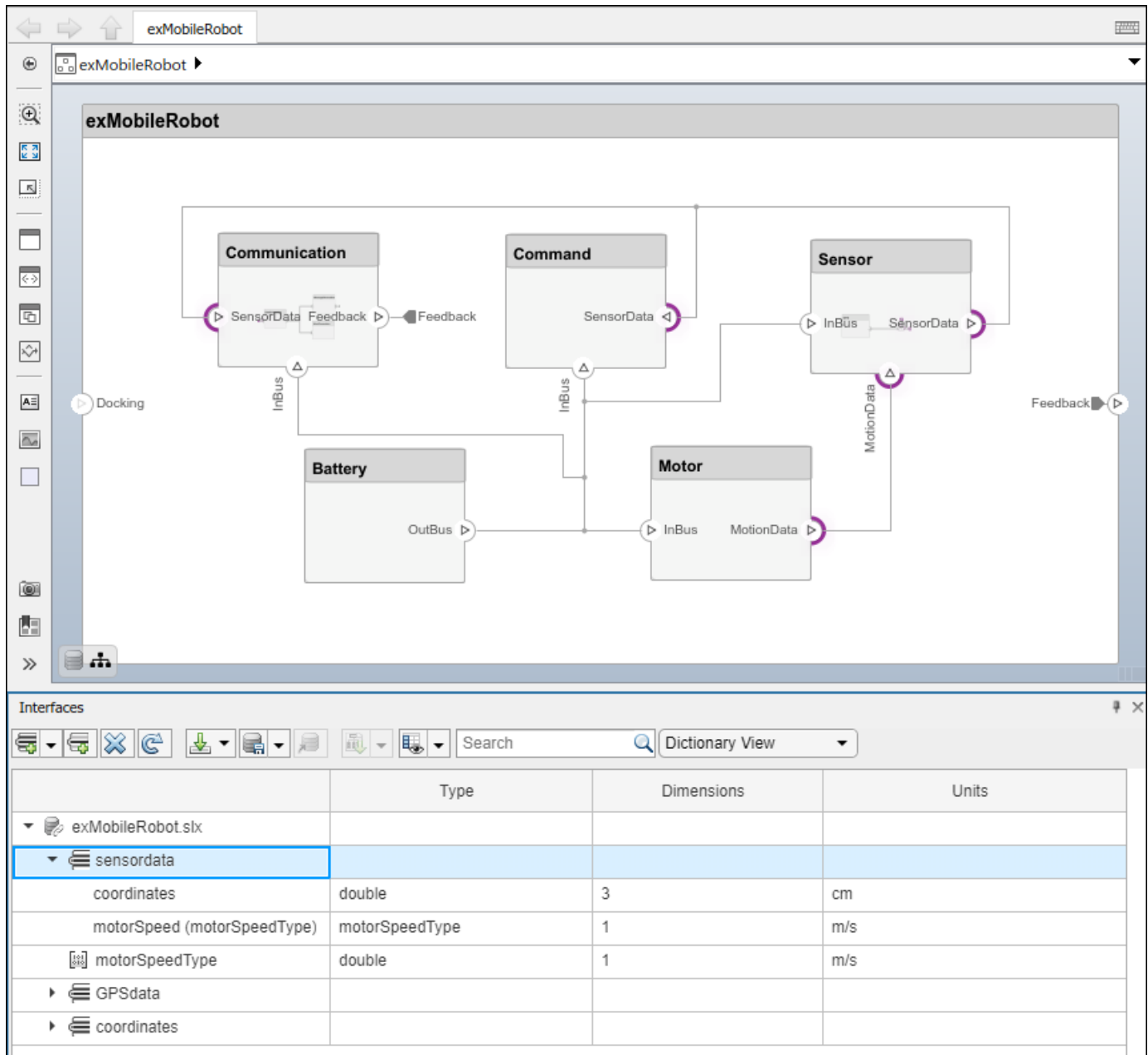
Make Owned Interfaces into Shared Interfaces

To convert an owned interface into a shared interface, right-click the port with the owned interface and select **Convert to shared interface**. Alternatively, use the `makeOwnedInterfaceShared` function.

Select Multiple Ports and Assign Data Interface

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

Highlight the ports that use a data interface definition by clicking the interface name in the **Interface Editor**.



Specify Source Element or Destination Element for Ports

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.

- 1 Create a component called `Motor` and connect it to the root architecture with ports named `MotionData` and `SpeedData`.
- 2 Define the data interface `Wheel` with the data elements `RotationSpeed` and `MaxSpeed`.
- 3 Assign the `Wheel` data interface to the ports on the connection.

- 4 Select the `MotionData` port name on the component. A dot and a list of data elements appear. From the list, select the source element `RotationSpeed`.
- 5 Assign the `MaxSpeed` destination element to the `SpeedData` port.

The screenshot shows a software development environment with a component diagram for 'exMobileRobot'. The diagram includes a 'Motor' component with two ports: 'SpeedData' and 'MotionData'. The 'SpeedData' port is connected to the 'MaxSpeed' element of the 'SpeedData' interface. The 'MotionData' port is connected to the 'RotationSpeed' element of the 'MotionData' interface. Below the diagram is an 'Interfaces' panel with a table listing the interfaces and their elements.

	Type	Dimensions	Units
exMobileRobot.slx			
Wheel			
RotationSpeed	double	1	m/s
MaxSpeed	double	1	m/s

See Also

Functions

connect | getDestinationElement | getSourceElement | createOwnedType | createInterface | makeOwnedInterfaceShared

Blocks

Component

More About

- “Create Interfaces” on page 3-5
- “Manage Interfaces with Data Dictionaries” on page 3-20
- “Interface Adapter” on page 3-16
- “Specify Physical Interfaces on Ports” on page 5-76
- “Modeling System Architecture of Small UAV” on page 1-32

Interface Adapter

A source port and its destination port may be defined by different data interfaces. Such a connection can represent an intermediate point in design, where components from different sources come together. To connect components with different data interfaces, use an Adapter block and the Interface Adapter. For interfaces terminology, see “Define Port Interfaces Between Components” on page 3-2.

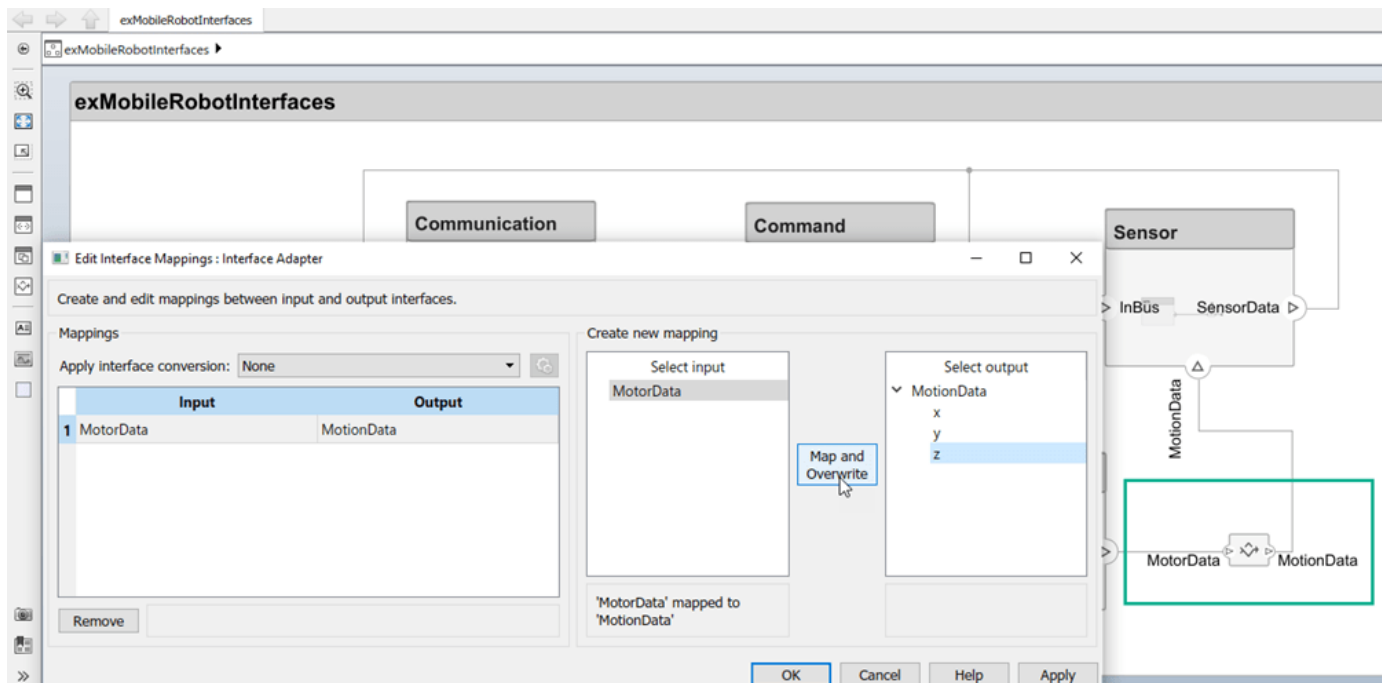
An *adapter* helps connect two components with incompatible port interfaces by mapping between the two interfaces. Use the Adapter block to implement an adapter. Open the Interface Adapter by double-clicking an Adapter block on the connection between the ports.

Use the Interface Adapter in System Composer™ to map interface elements between two ports. You can also use the Interface Adapter to apply an interface conversion that breaks algebraic loops with unit delays, or insert a rate transition for different sample time rates. When output interfaces are undefined, you can use input interfaces in bus creation mode of the Interface Adapter to author owned output interfaces.

Map Incompatible Interfaces

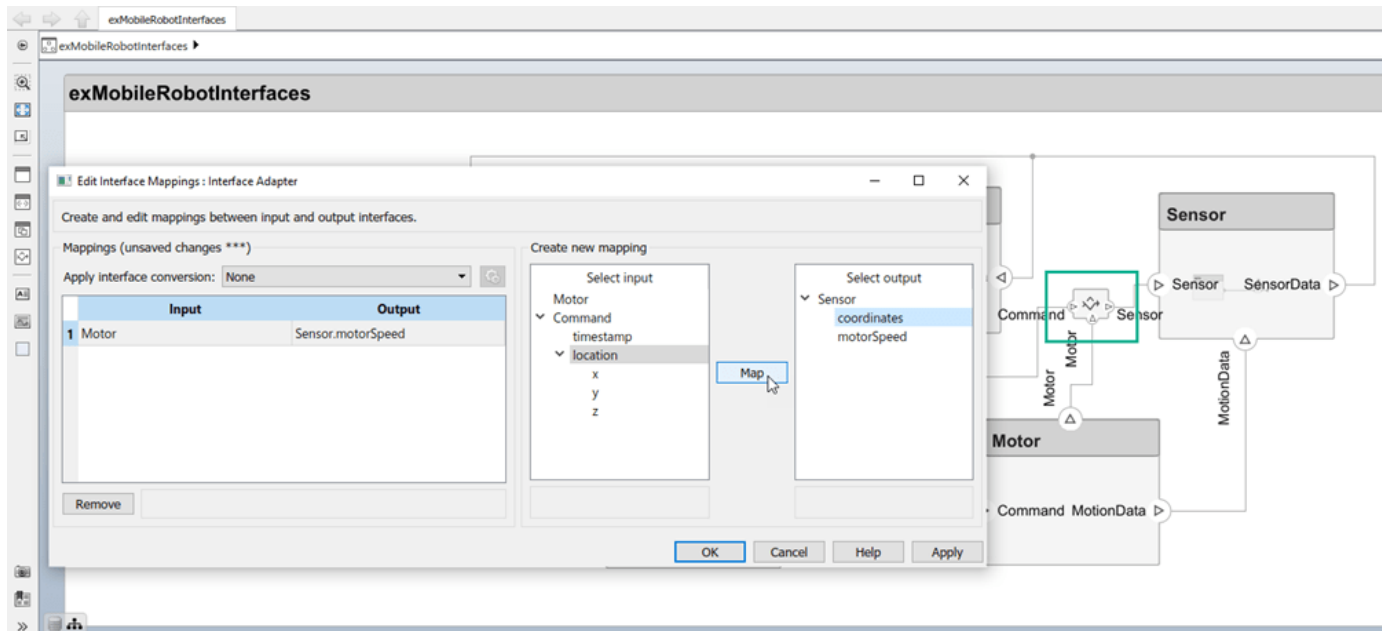
When two connected components with Simulink® behaviors have incompatible interfaces, use an Adapter block and the Interface Adapter to define the port connections.

- 1 Add an Adapter block 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 a data element. In the **Select output** box, select a data element.
- 4 Click the **Map and Overwrite** button.



You can use an Adapter block to map similar interfaces for an N:1 connection, which is an Adapter with more than one input port and a single output port. A data element from each input connection maps to the output connection data elements.

Change the number of input ports on an Adapter block in the same way you add and remove component ports. For more information, see “Compose Architecture Visually” on page 1-2.



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 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 the components 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 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.

Use Bus Creation Mode to Author Owned Interfaces

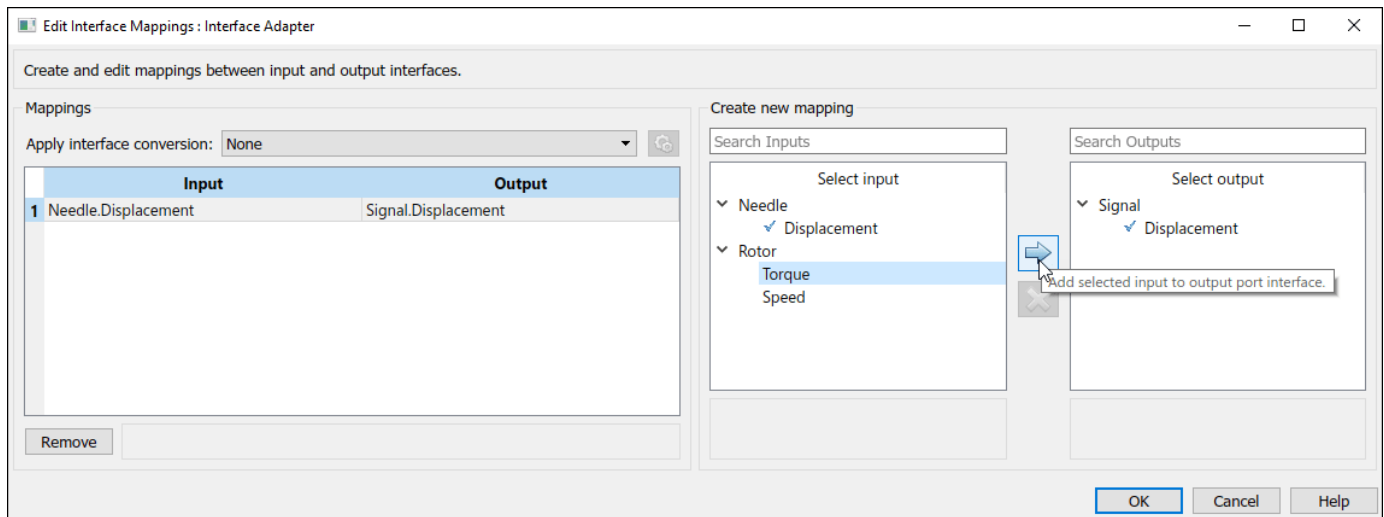
When input ports for an Adapter block are typed by interfaces from incoming connections and no interfaces are defined on the output ports of the Adapter, you can use these interface elements to


author owned interfaces for outgoing connections. Instead of pre-defining interface structures, you can create the bus structure.

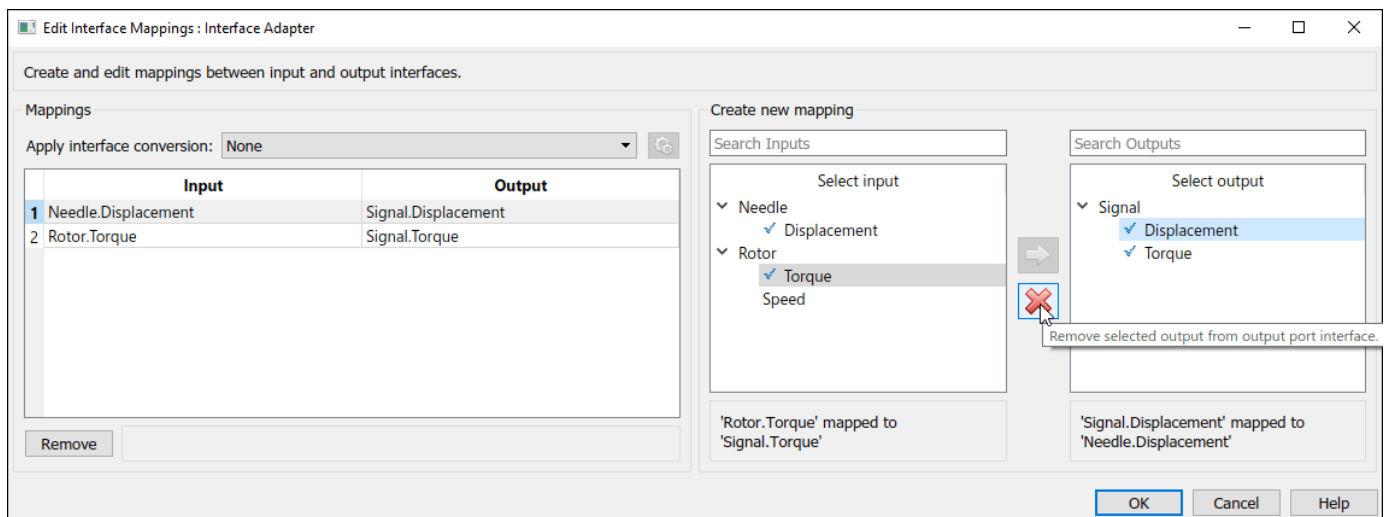
Double-click the Adapter block to open the Interface Adapter dialog in bus creation mode. Click the



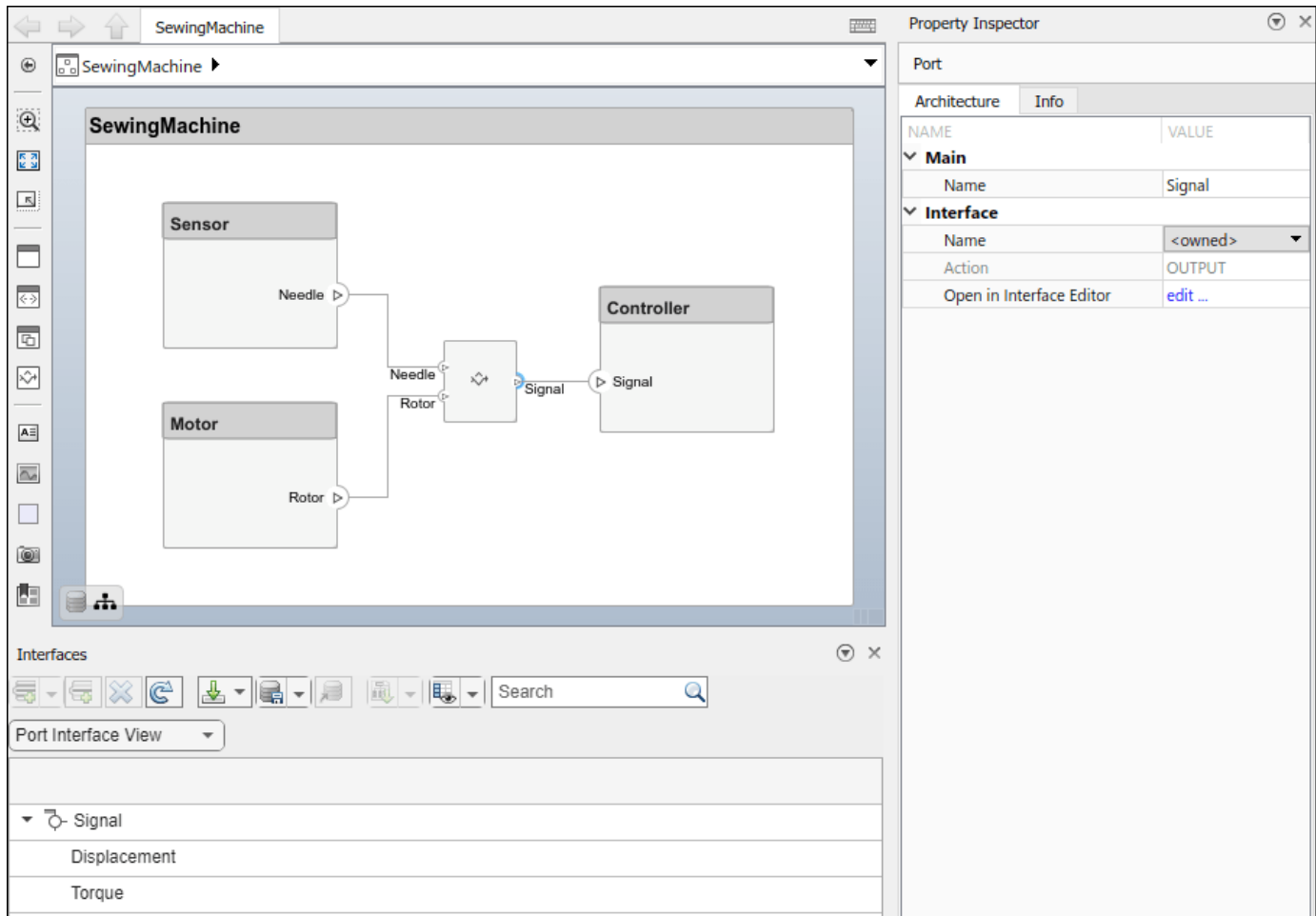
button to add an input data interface, data element, or value type to the output port interface.



Click the  button to remove an output data interface, data element, or value type from the output port interface.



Click **OK** to apply the changes.



The owned interface on the output port of the Adapter block propagates to the connected input port `Signal` on the Controller component.

To convert an owned interface into a shared interface, right-click the port with the owned interface and select `Convert to shared interface`.

See Also

Blocks

Adapter | `makeOwnedInterfaceShared`

More About

- “Create Interfaces” on page 3-5
- “Assign Interfaces to Ports” on page 3-10
- “Manage Interfaces with Data Dictionaries” on page 3-20
- “Define Port Interfaces Between Components” on page 3-2
- “Merge Message Lines Using Adapter Block” on page 8-32

Manage Interfaces with Data Dictionaries

In this section...

“Mobile Robot Architecture Model with Interfaces” on page 3-20

“Save, Link, and Delete Interfaces” on page 3-20

Engineering systems often share interface definitions across multiple components or subsystems. Data interfaces in System Composer can be stored either locally in a model or in a data dictionary, depending on the maturity of your system. For interfaces terminology, see “Define Port Interfaces Between Components” on page 3-2.

An interface data dictionary is a consolidated list of all the interfaces and value types in an architecture and where they are used.

Local interfaces on a System Composer model can be saved in an interface data dictionary using the **Interface Editor**. Interface dictionaries can be reused between models that need to use a given set of interfaces, elements, and value types. Data dictionaries are stored in separate SLDD files.


For more advanced dictionary referencing techniques, see “Reference Data Dictionaries” on page 3-23.

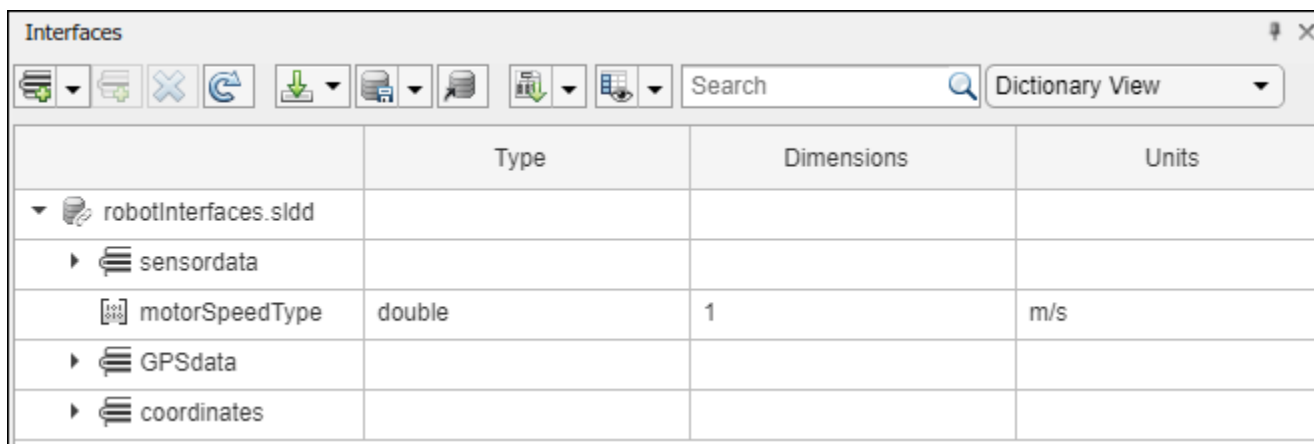
Mobile Robot Architecture Model with Interfaces

This example shows a mobile robot hardware architecture with interfaces defined.

Save, Link, and Delete Interfaces


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 a data interface to a new or existing data dictionary. To create a new data dictionary, select **Save to new dictionary**. Provide a dictionary name.



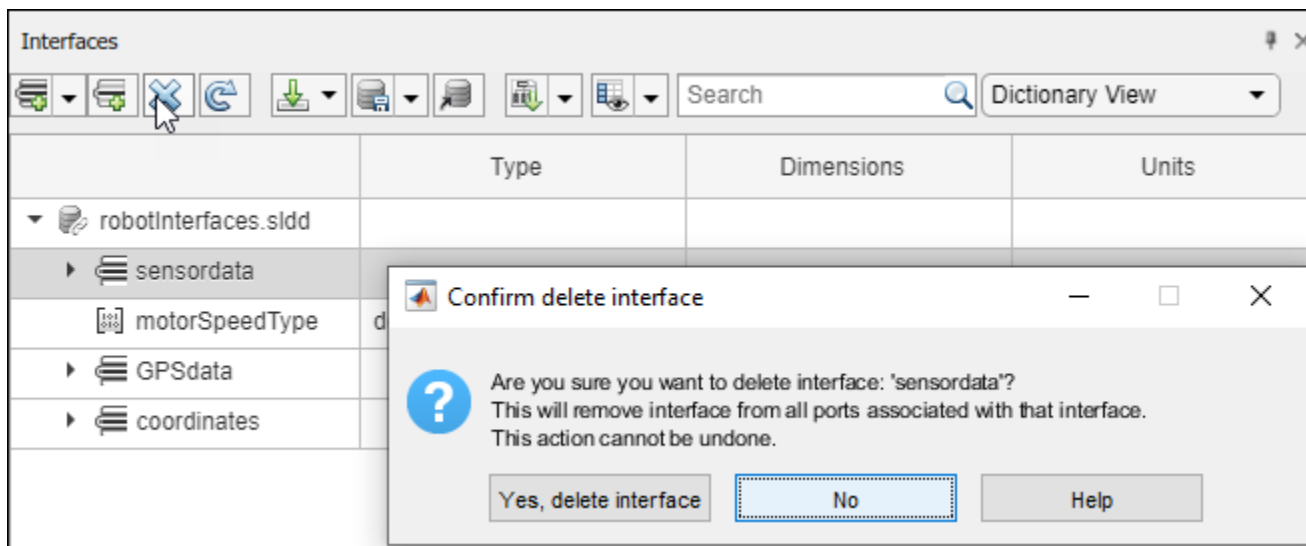
	Type	Dimensions	Units
robotInterfaces.sldd			
sensordata			
motorSpeedType	double	1	m/s
GPSdata			
coordinates			


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 a data interface from a dictionary using the  button. If the data interface is already being used by ports in a currently open model, the software returns a warning message. The data interface is then removed from any ports in the open model that are associated with the data interface.

If a data 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.



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 .slidd 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](#) | [openDictionary](#) | [saveToDictionary](#) | [linkDictionary](#) | [unlinkDictionary](#) | [makeOwnedInterfaceShared](#)

More About

- “Create Interfaces” on page 3-5
- “Assign Interfaces to Ports” on page 3-10
- “Interface Adapter” on page 3-16

- “Reference Data Dictionaries” on page 3-23
- “Specify Physical Interfaces on Ports” on page 5-76

Reference Data Dictionaries

In this section...


“Add Referenced Data Dictionaries” on page 3-23

“Use Referenced Data Dictionaries for Projects with Multiple Models” on page 3-24

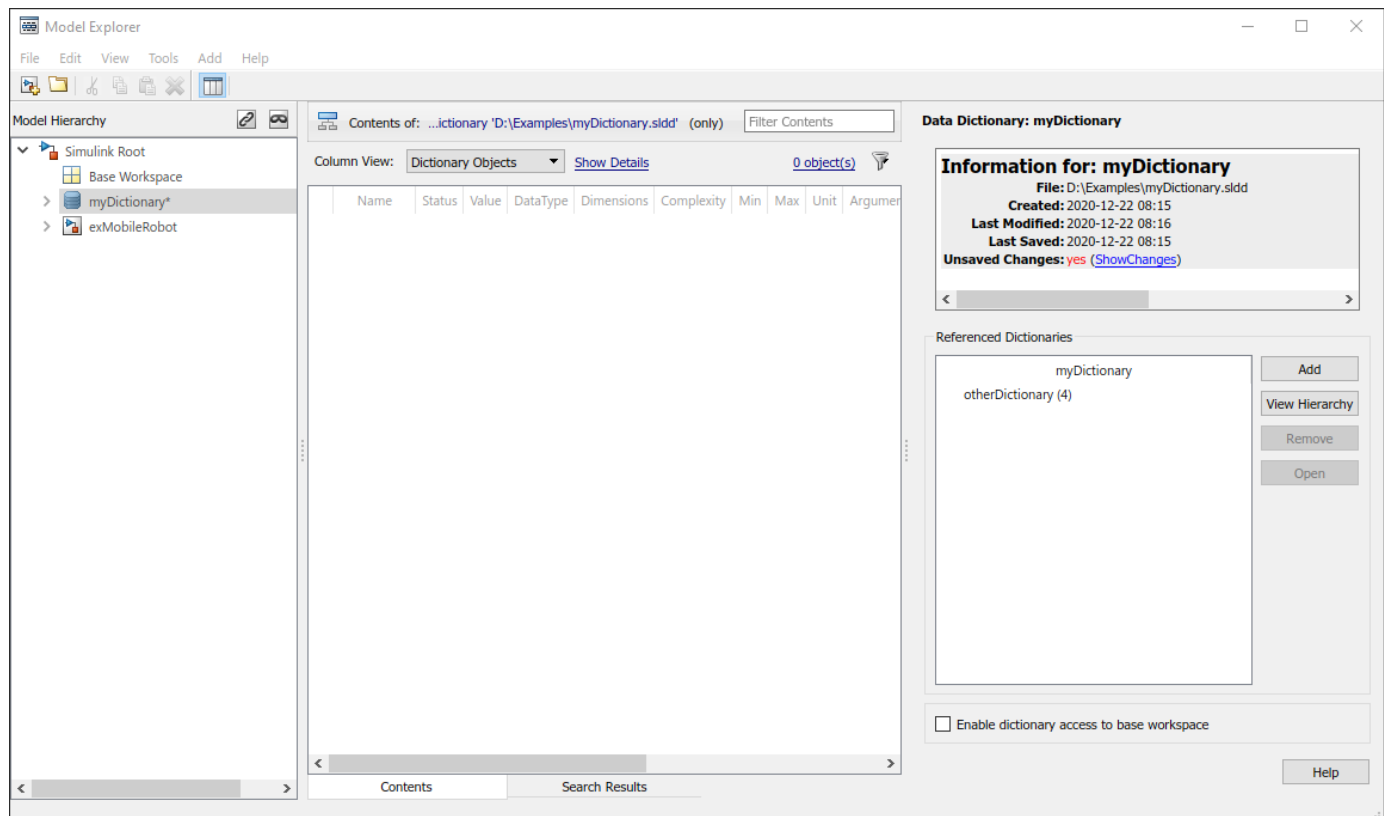
Referenced dictionaries in System Composer may be useful when multiple models need to share some, but not all, interface definitions. and to allow communication between the models. A data dictionary can reference one or more other data dictionaries. The interface definitions in the referenced dictionaries are visible in the parent dictionary and can be used by a model that is linked to the parent dictionary. For interfaces terminology, see “Define Port Interfaces Between Components” on page 3-2.

To create a data dictionary from interfaces in a model dictionary, see “Manage Interfaces with Data Dictionaries” on page 3-20.

Add Referenced Data Dictionaries

To add a dictionary reference, open the Model Explorer by clicking , or by navigating to **Modeling > Design > Model Explorer**.

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**.



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							
☰ OtherInterface1							
☰ OtherInterface2							

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

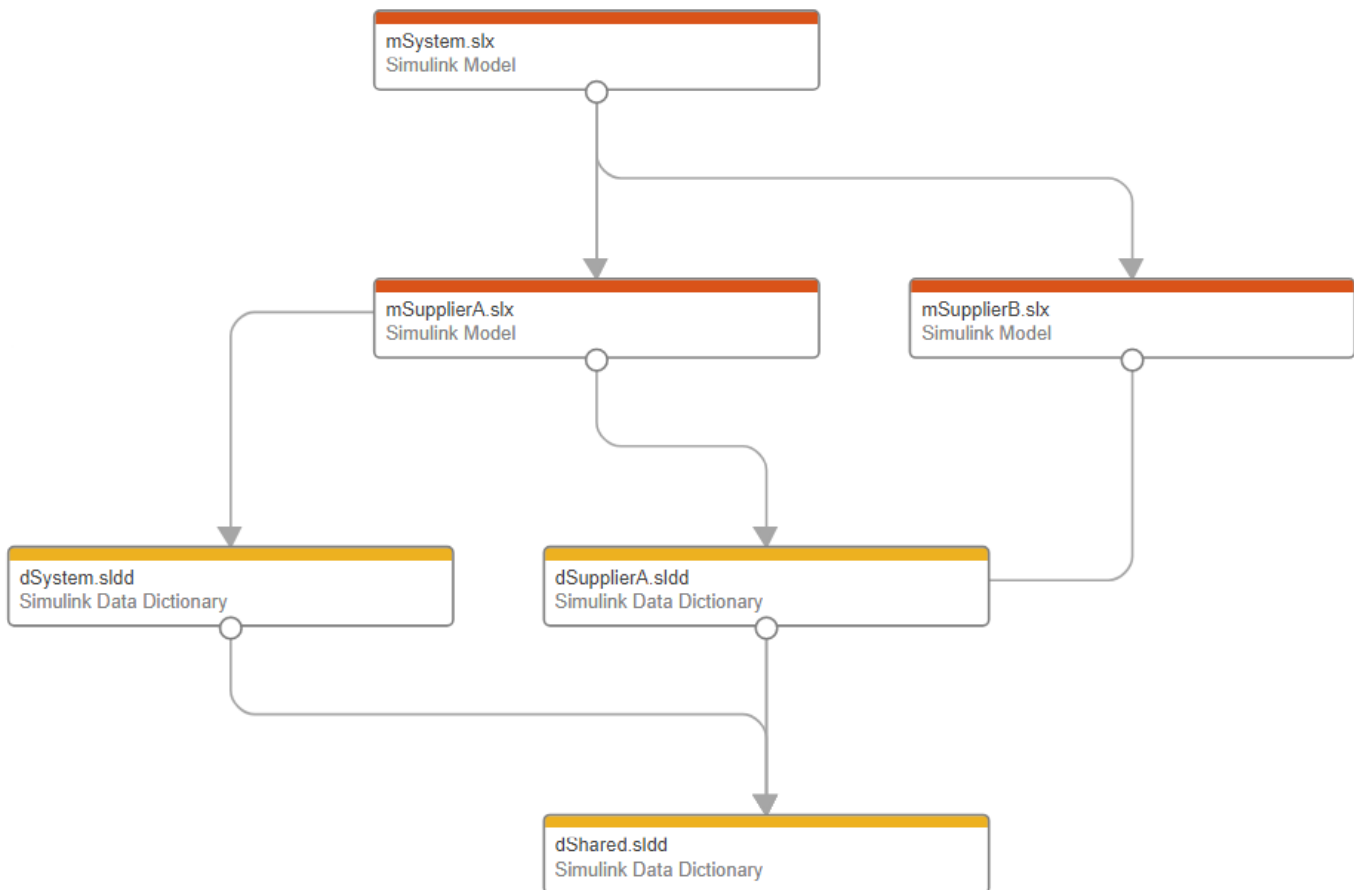
Note Referenced dictionaries can reference other data dictionaries. A model that links to a dictionary has access to all interface definitions in referenced dictionaries, including indirectly referenced dictionaries.

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. For more information, see "Organize System Composer Files in Projects" on page 10-2,

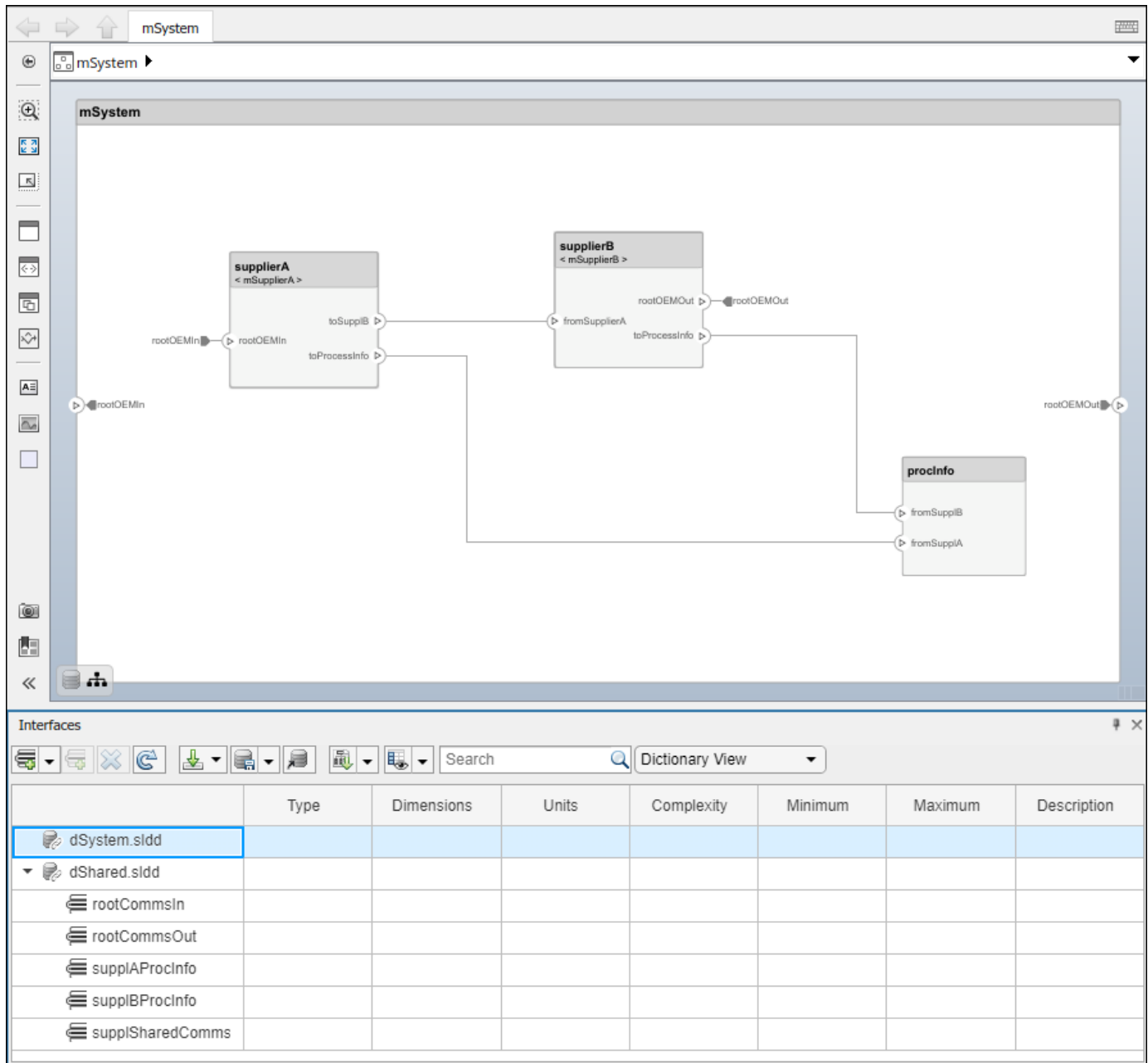
At the same time, each model may have interface definitions that are relevant only to its internal operations. For example, different components of a system may be represented by different models, with different teams or different suppliers working on each model, with a system integrator working on the "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.

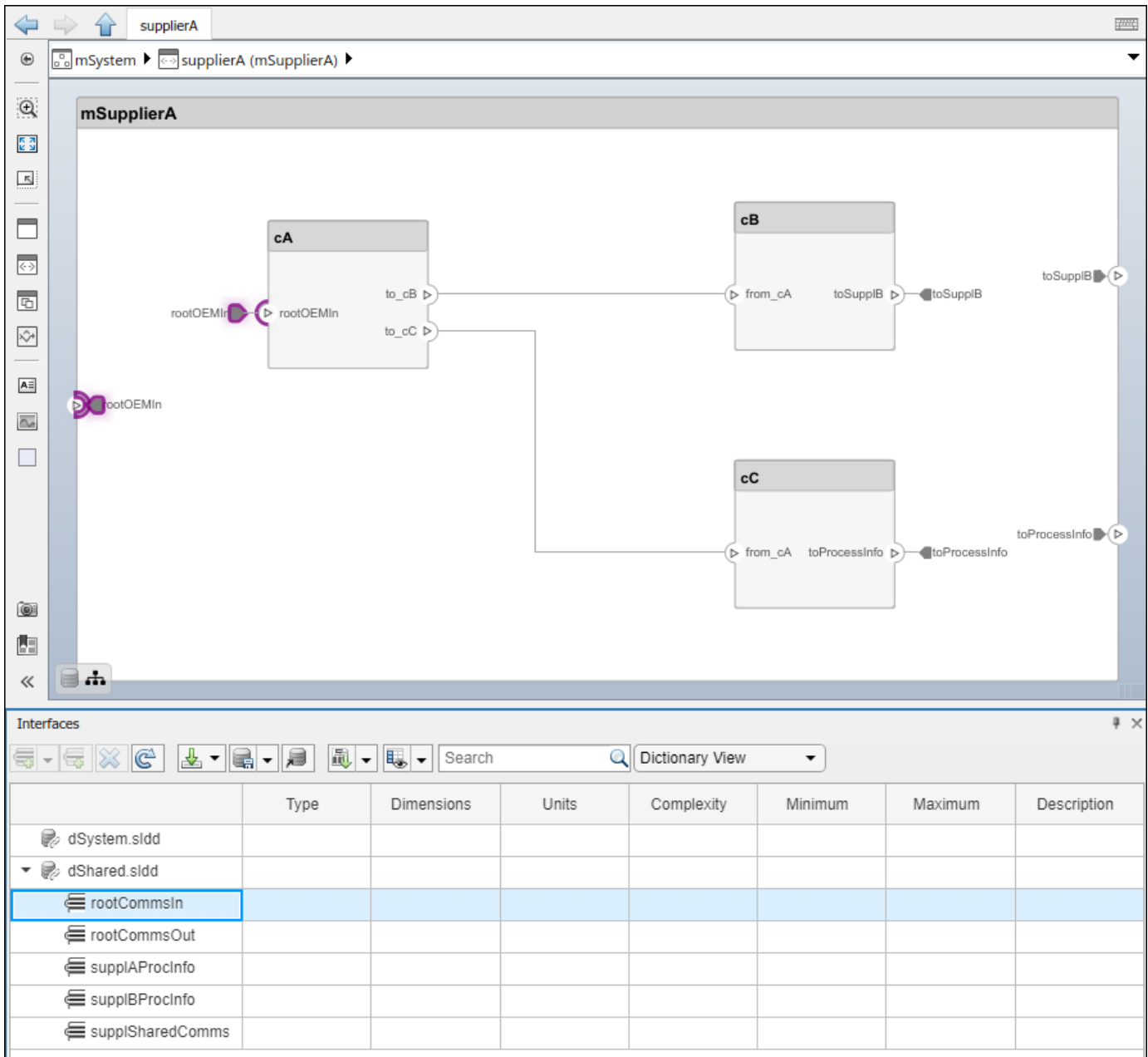


The above diagram depicts a project with three models. The model `mSystem.slx` represents a system integration model, and `mSupplierA.slx` and `mSupplierB.slx` represent supplier models. The data dictionary `dShared.sldd` contains interface definitions shared by all the models. The system integration 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.



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.



See Also

[addReference](#) | [removeReference](#)

More About

- “Create Interfaces” on page 3-5
- “Assign Interfaces to Ports” on page 3-10
- “Manage Interfaces with Data Dictionaries” on page 3-20
- “Specify Physical Interfaces on Ports” on page 5-76

- “Organize System Composer Files in Projects” on page 10-2

Define Architectural Properties

- “Define Profiles and Stereotypes” on page 4-2
- “Use Stereotypes and Profiles” on page 4-10
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22
- “Organize and Link Requirements” on page 4-24
- “Design Architecture Models” on page 4-27
- “Define Stereotypes and Perform Analysis” on page 4-34
- “Simulate Architectural Behavior” on page 4-43

Define Profiles and Stereotypes

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

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.

A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architecture language elements by adding domain-specific metadata.

Apply stereotypes to model elements such as root-level architecture, component architecture, connectors, ports, data interfaces, value types, and functions. Functions only apply to software architectures. A model element can have multiple stereotypes. Stereotypes provide model elements with a common set of property fields, such as mass, cost, and power.

A property is a field in a stereotype. You can specify property values for each element to which the stereotype is applied.

Use properties to store quantitative characteristics, such as weight or speed, that are associated with a model element. Properties can also be descriptive or represent a status. You can view and edit the properties of each element in the architecture model using the Property Inspector.

Open the Property Inspector by navigating to **Modeling > Property Inspector**.

A profile is a package of stereotypes to create a self-consistent domain of element types.

Author profiles and apply profiles to a model using the **Profile Editor**. You can store stereotypes for a project in one profile or in several. When you save profiles, they are stored in XML files.

In this topic, you will learn how to:

- 1 Create a profile and define stereotypes with properties.
- 2 Define default stereotypes in a profile to be added to any new element in a model with that applied profile.
- 3 Use stereotype-based styling that enhances the appearance of the model based upon specific features each element represents.

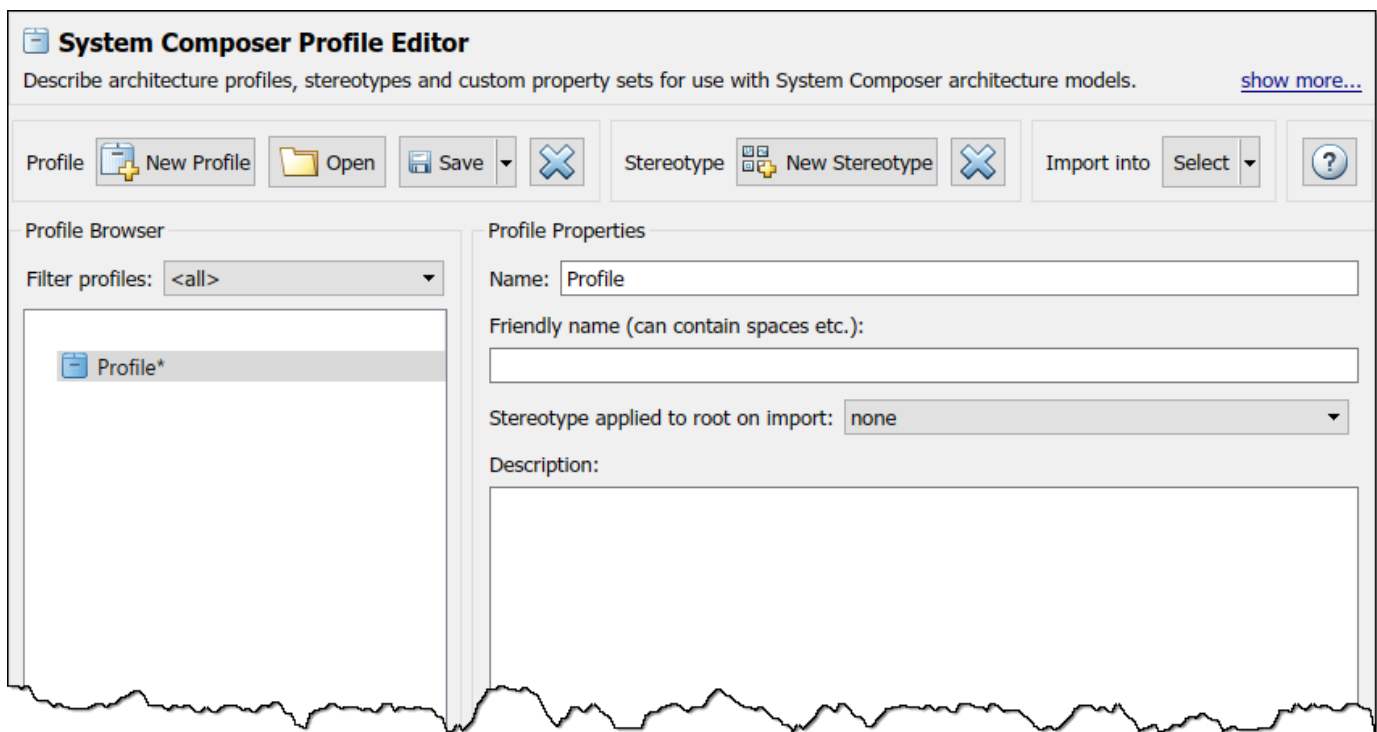
Create a Profile and Add Stereotypes


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** by navigating to **Modeling > Profile Editor**. Click **New Profile**. Select the 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:

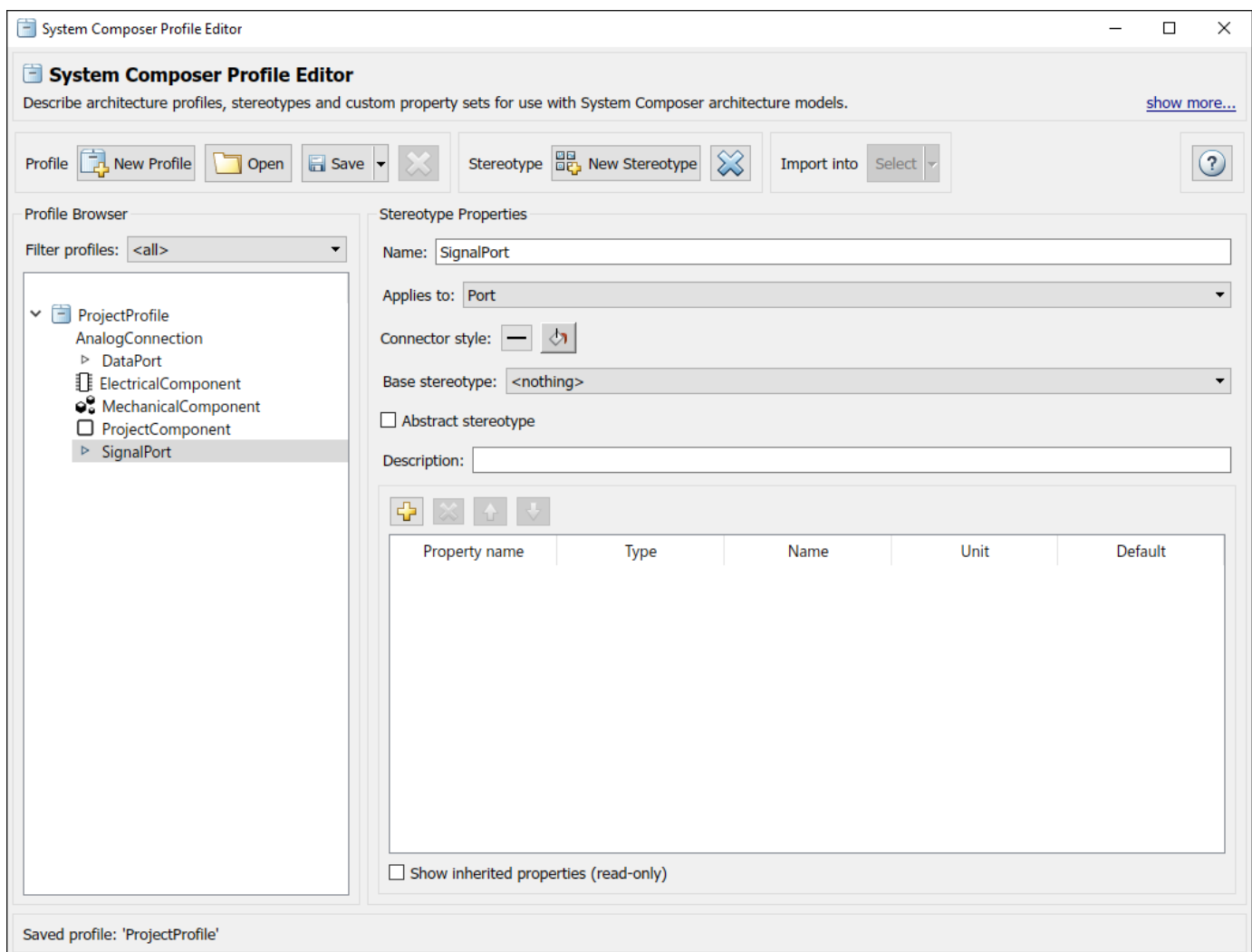
- **Name** — The name of the stereotype, for example, `ElectricalComponent`.
- **Applies to** — The model element type to which the stereotype applies. This option can be `<all>`, `Component`, `Port`, `Connector`, `Interface`, or `Function`. You can apply this stereotype only to a model element of this type. The model element type `Function` is only available for software


architecture models. For more information, see “Apply Stereotypes to Functions of Software Architectures” on page 8-29.

- **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 option 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



Add, delete, and reorder properties using the property toolbar: 

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:

Applies to:

Base stereotype:

Abstract stereotype

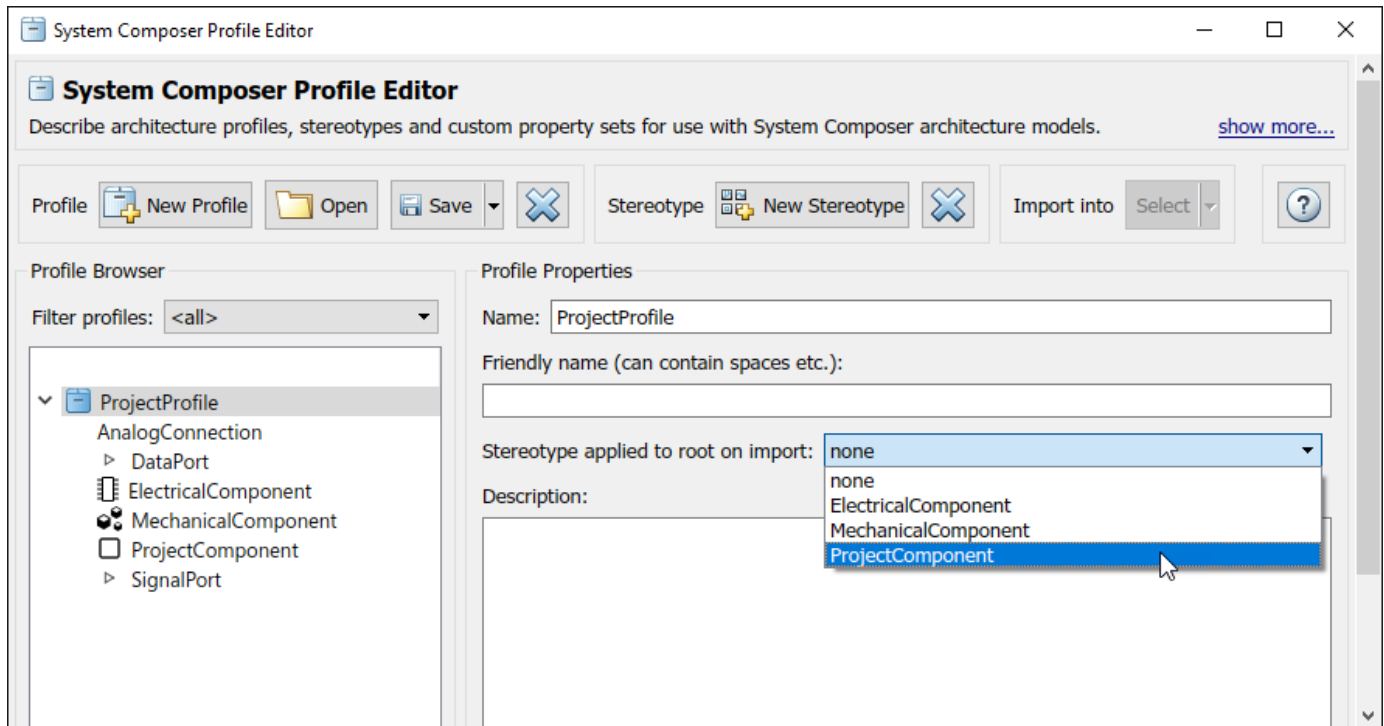
Description:



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

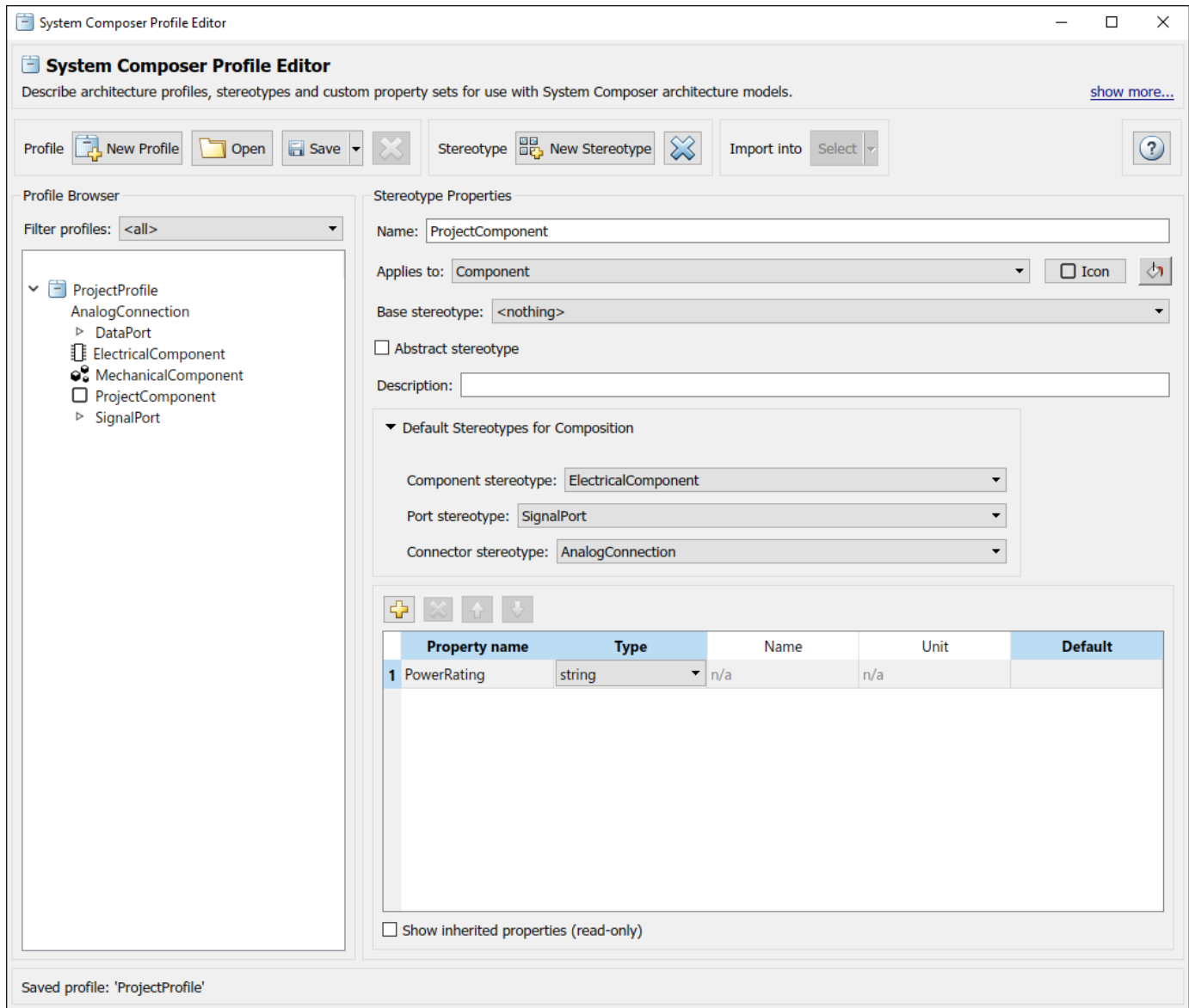
Define Default Stereotypes

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 as **ProjectComponent** in the **Profile Editor** using the **Stereotype applied to root on import** field.



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 a project component to be analog connections, set `AnalogConnection` as a default stereotype for the `ProjectComponent` stereotype.



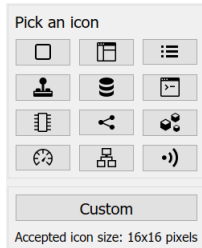
When you import the profile ProjectProfile into a model:

- The ProjectComponent stereotype is automatically applied to the root architecture.
- The ElectricalComponent stereotype is automatically applied to all new components in the architecture model.
- The SignalPort stereotype is automatically applied to all new ports.
- The AnalogConnection stereotype is automatically applied to all new connections.

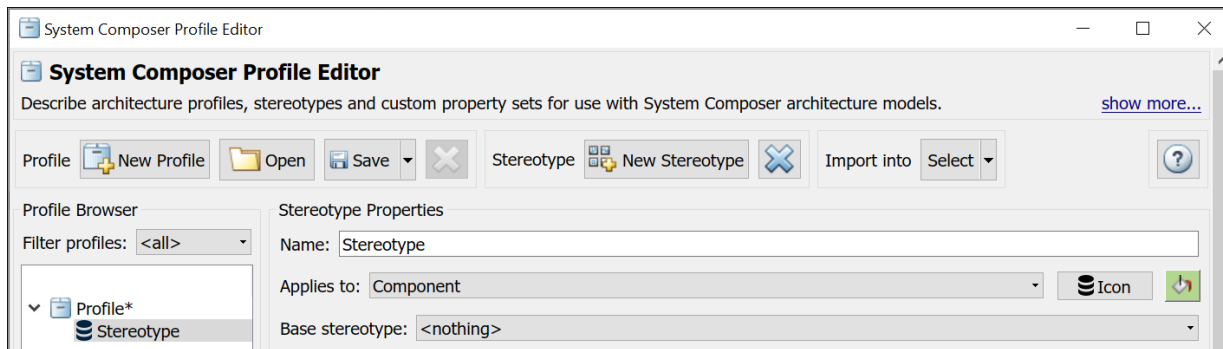
Use Stereotype-Based Styling

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.

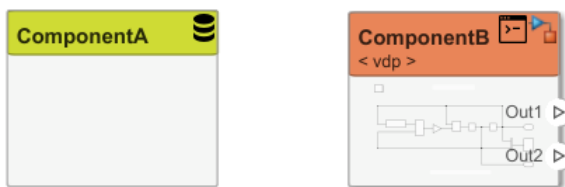
You can use provided icons for the component stereotypes or use your own custom icon images. Custom icons support .png, .jpeg, or .svg image files of size 16-by-16 pixels. The custom icons are displayed as badges on the components for which the stereotypes are applied.




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



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.

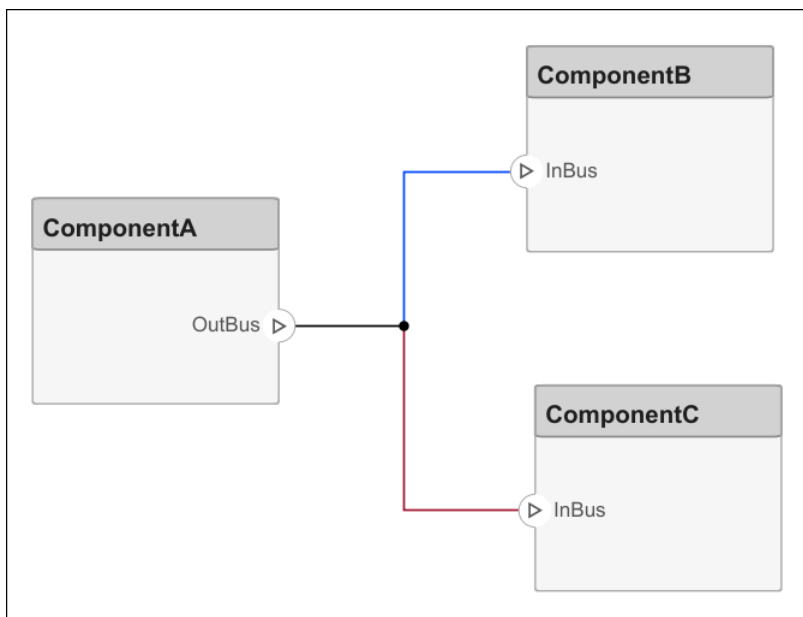


Similarly, you can style architecture connectors using the stereotype settings. You can style connectors by using connector, port, or port interface stereotypes. Customize styling provides various color and line style choices. Connector styles are also reflected in architecture and spotlight views.

Stereotype Properties	
Name:	<input type="text" value="Stereotype"/>
Applies to:	<input type="text" value="Connector"/>
Connector style:	<input type="text" value="..."/> 
Base stereotype:	<input type="text" value="<nothing>"/>
<input type="checkbox"/> Abstract stereotype	
Description:	<input type="text"/>

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

[hasStereotype](#) | [hasProperty](#) | [editor](#) | [systemcomposer.profile.Profile](#) | [systemcomposer.profile.Property](#) | [systemcomposer.profile.Stereotype](#)

More About

- “Use Stereotypes and Profiles” on page 4-10
- “Analyze Architecture” on page 7-2
- “Analysis Function Constructs” on page 7-9
- “Modeling System Architecture of Small UAV” on page 1-32
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

Use Stereotypes and Profiles


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

In this topic, you will learn how to:

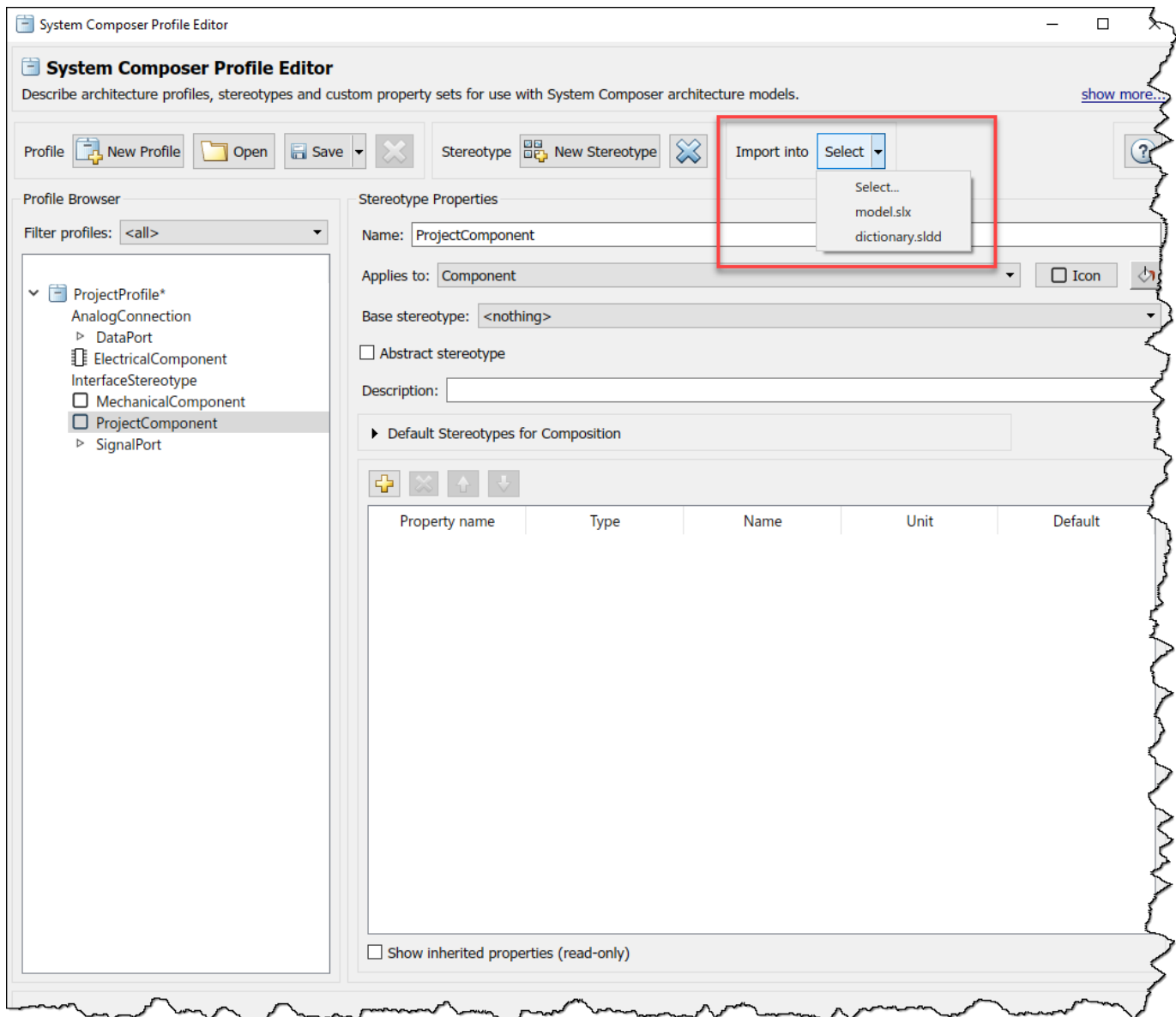
- 1 Import profiles into a model or a dictionary.
- 2 Apply a stereotype to a model element and add property values.
- 3 Remove stereotypes using the Property Inspector.
- 4 Extend stereotypes with other stereotypes to include their properties through an inherited mechanism. For example, a `UserInterface` stereotype can be an extension of a `SoftwareComponent` stereotype, and add a property called `ScreenResolution`.

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. Navigate to **Modeling >**

Profiles > Import . Select the profile to import. An architecture model can use multiple profiles at once.

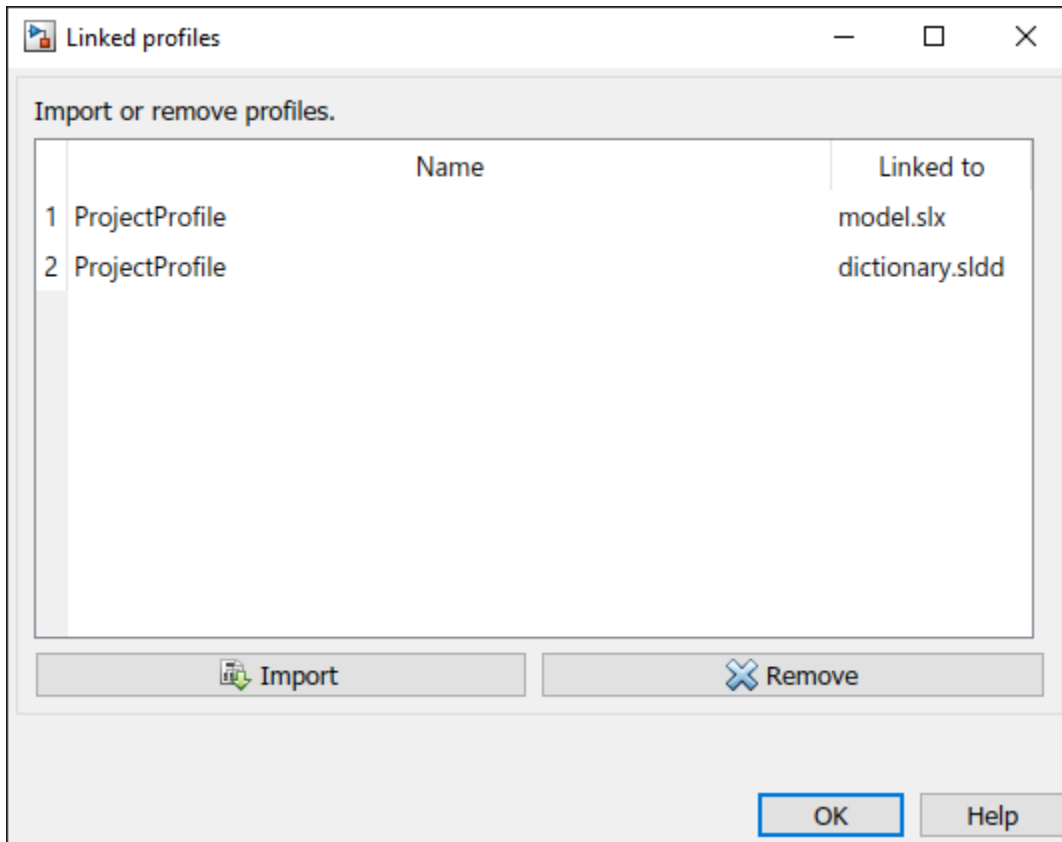
Alternatively, open the **Profile Editor** by navigating to **Modeling > Profile Editor**. You can import a profile into any open dictionaries or models.



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, navigate to **Modeling > Profiles > Manage**



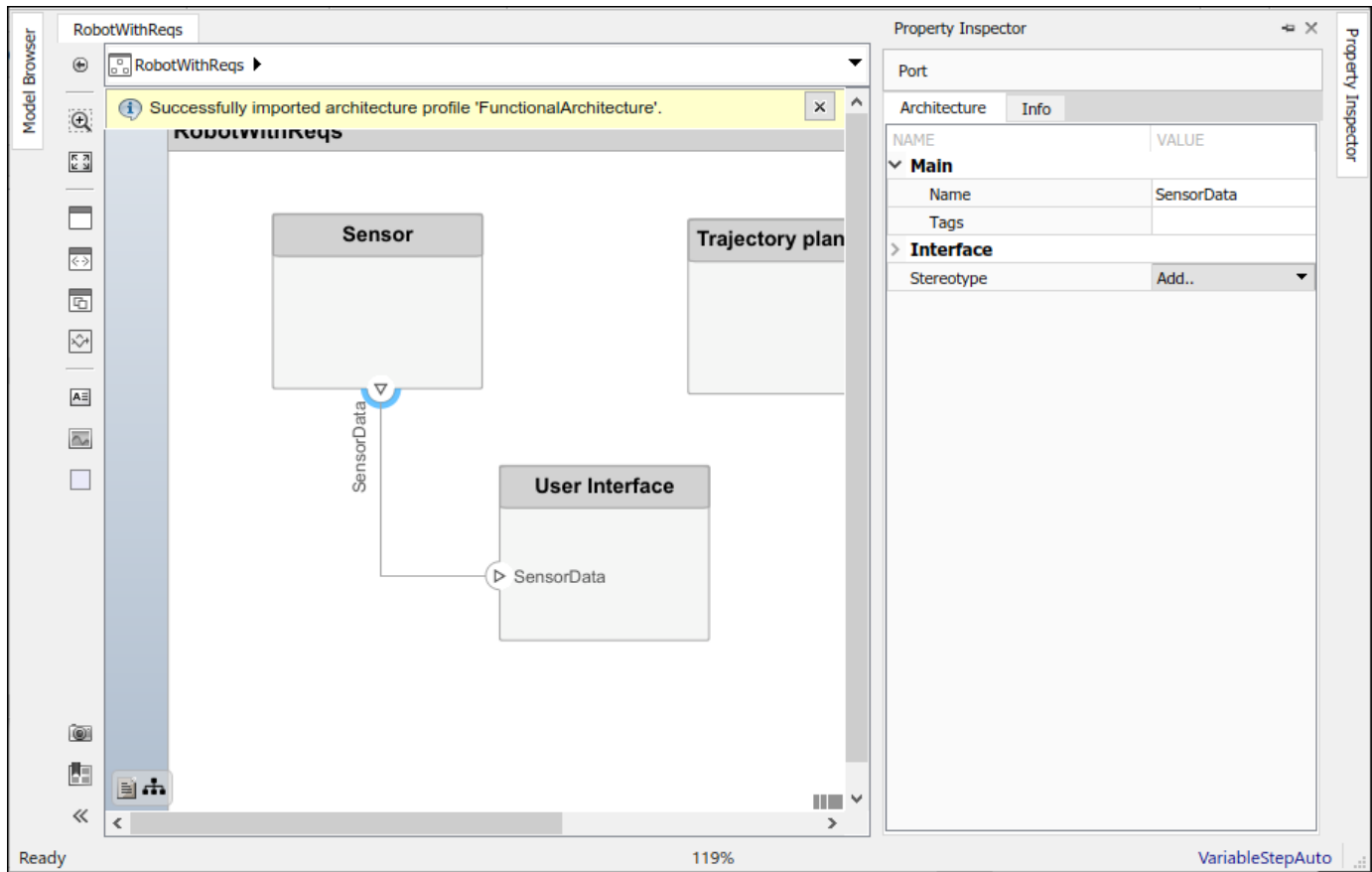


Apply Stereotypes

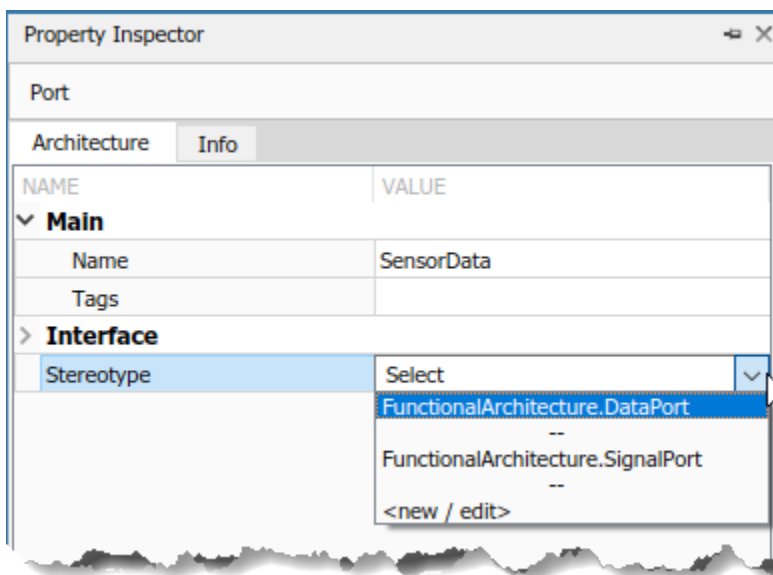
Apply stereotypes to architecture model elements using the Property Inspector or the Apply Stereotypes dialog. You can also quick-insert a new component with the stereotype applied. For information about applying stereotypes to functions in software architectures, see “Apply Stereotypes to Functions of Software Architectures” on page 8-29.

Apply a Stereotype using the Property Inspector

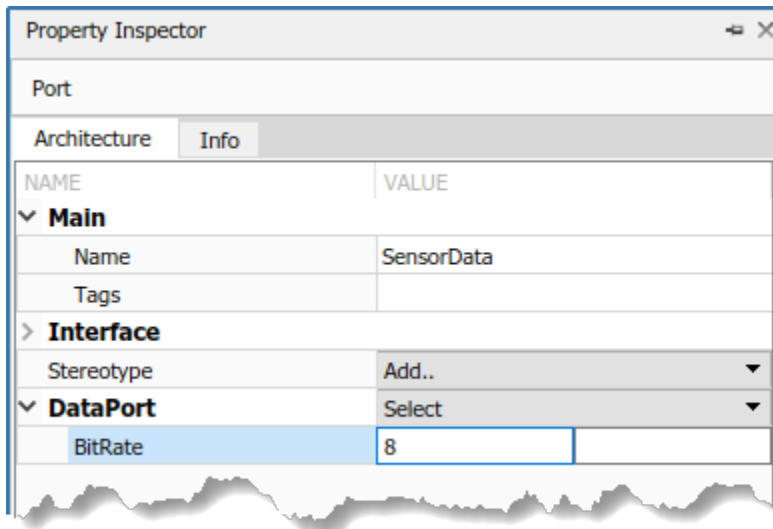
Once the profile is available in the model, open the Property Inspector by navigating to **Modeling > Property Inspector**. Select a model element.



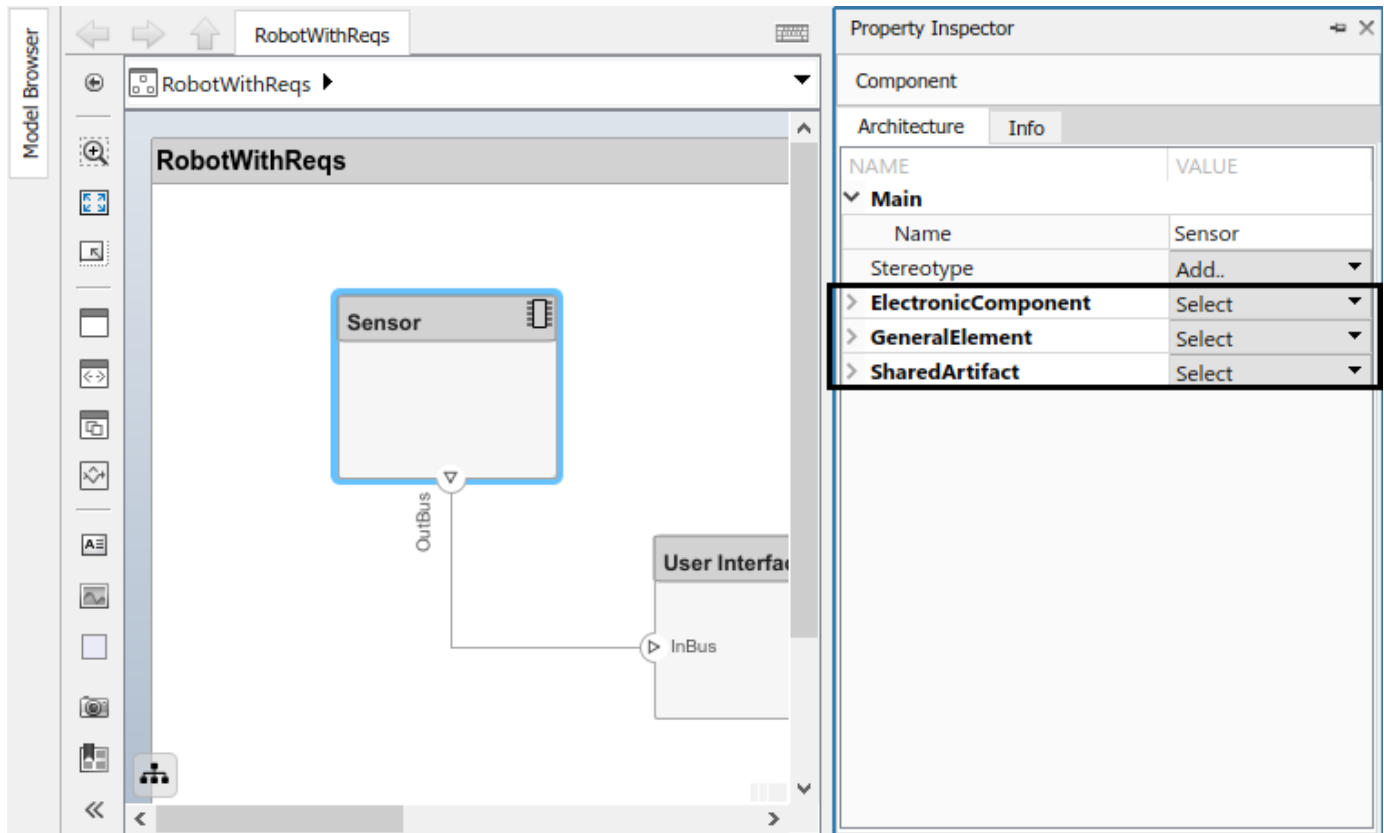
In the **Stereotype** field, use the drop-down to select the stereotype. Only the stereotypes that apply to the current element type (for example, a port) are available for selection. If no stereotype exists, you can use the **<new / edit>** option to open the **Profile Editor** and create one.



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.



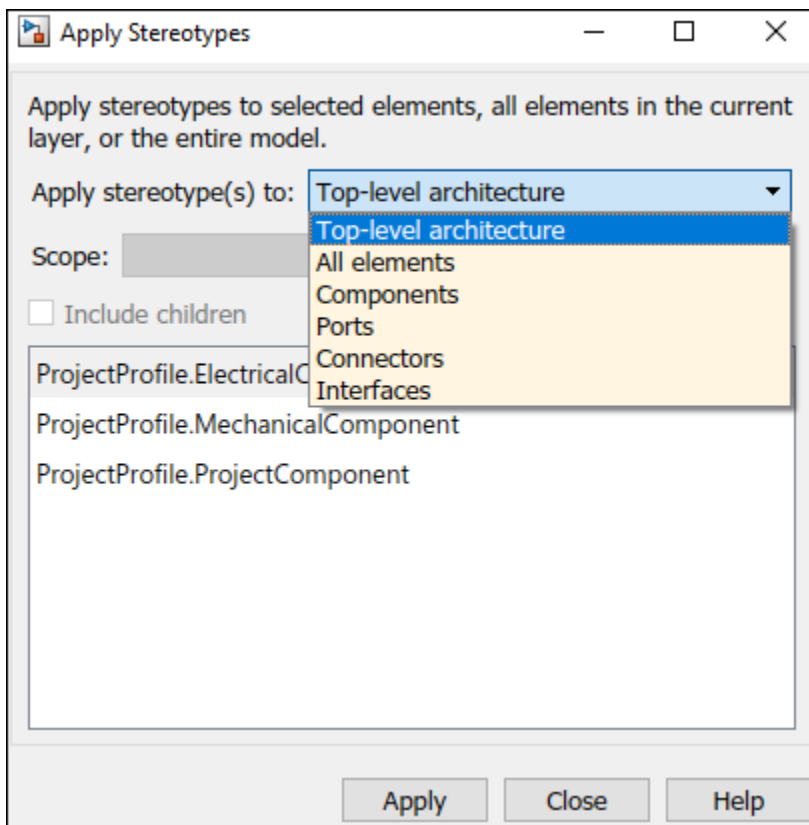
You can set multiple stereotypes for each element.



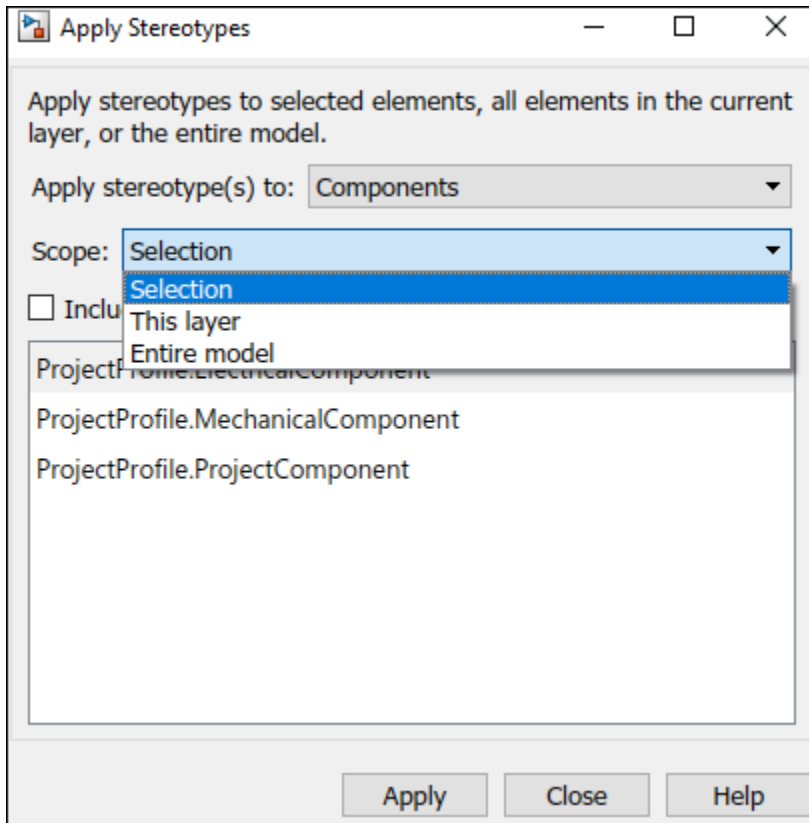
Use Apply Stereotypes Dialog to Batch Apply Stereotypes

You can also apply component, port, connector, and interface stereotypes to all applicable elements at the same architecture level. Navigate to **Modeling > Apply Stereotypes**. In Apply Stereotypes, from **Apply stereotype(s) to**, select Top-level architecture, All elements, Components, Ports, Connectors, or Interfaces.

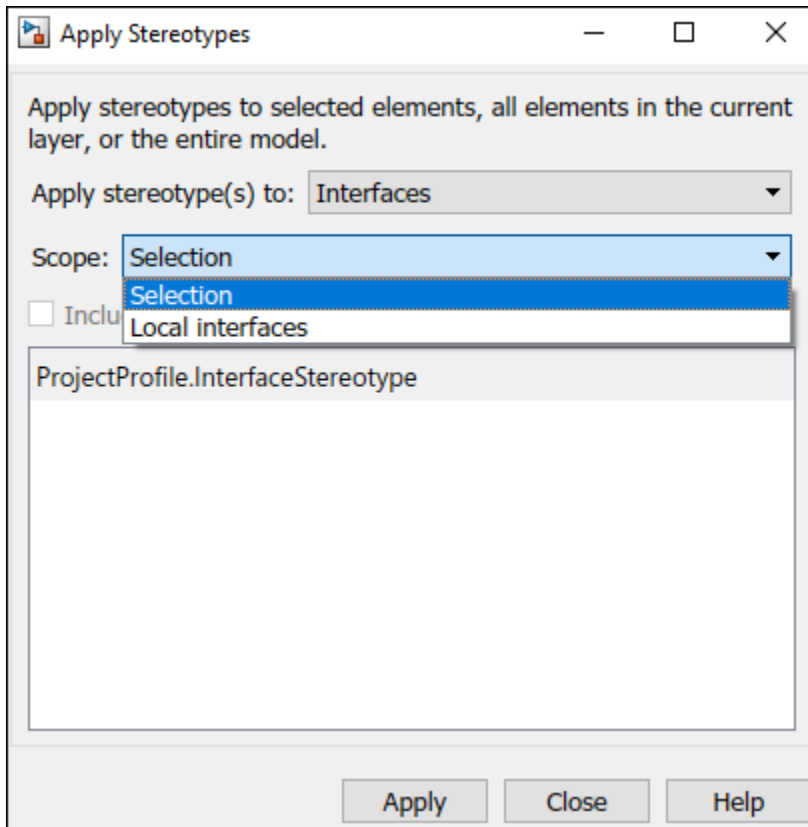
Note The Interfaces option is only available if interfaces are defined in the **Interface Editor**. For more information, see “Create Interfaces” on page 3-5.



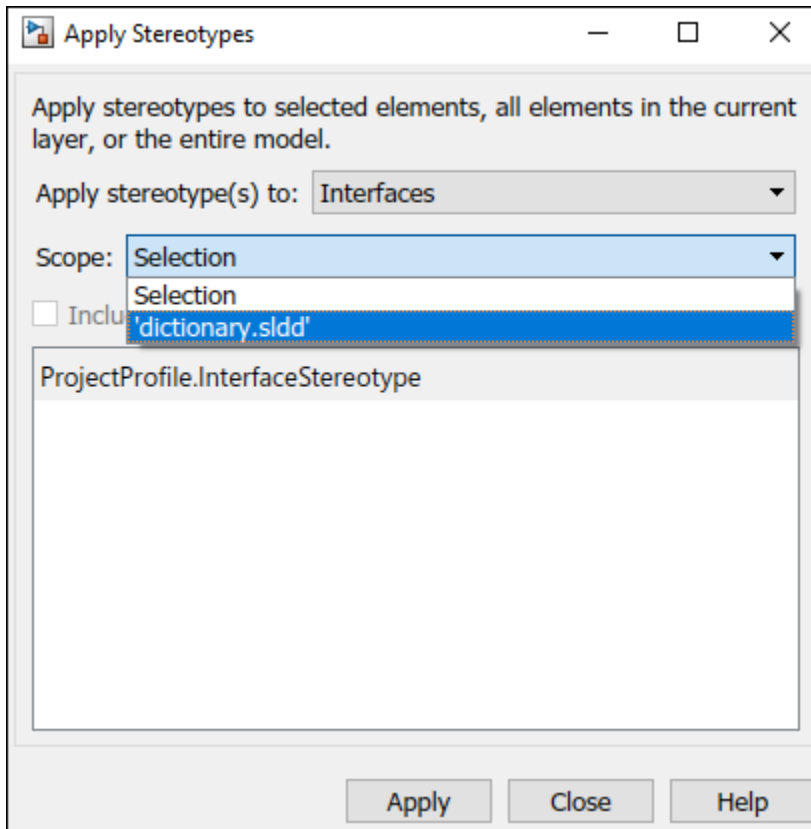
You can also apply stereotypes by selecting a single model element. From **Scope**, select Selection, This layer, or Entire model.



You can also apply stereotypes to data interfaces or value types. 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**.



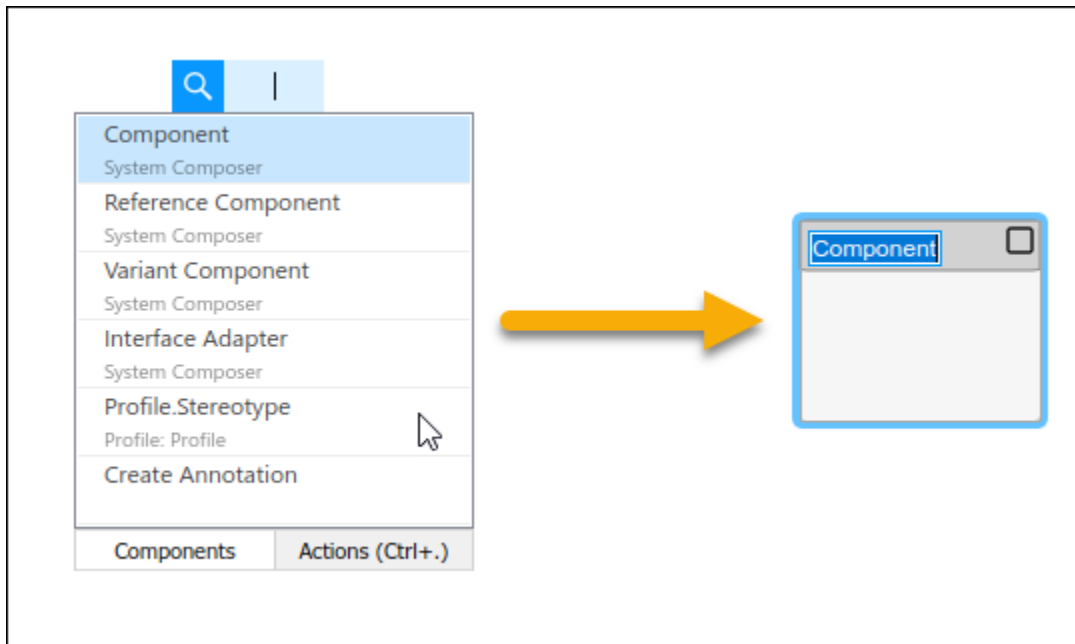
When interfaces are stored and shared across a data dictionary and you select one or more interfaces in the **Interface Editor**, the options for **Scope** are **Selection** and either `dictionary.sldd` or the name of the dictionary currently in use.



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.

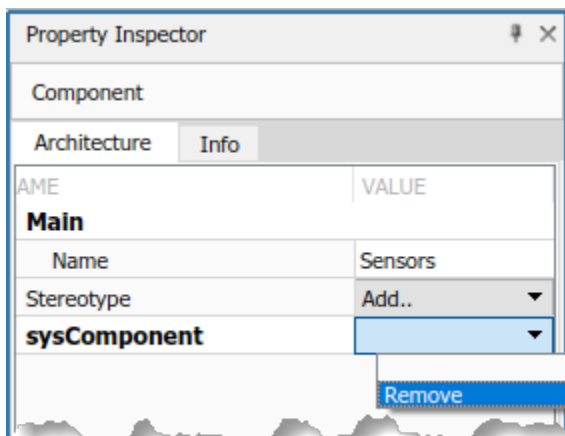
Quick-Insert New Component with Stereotype Applied

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 Stereotypes

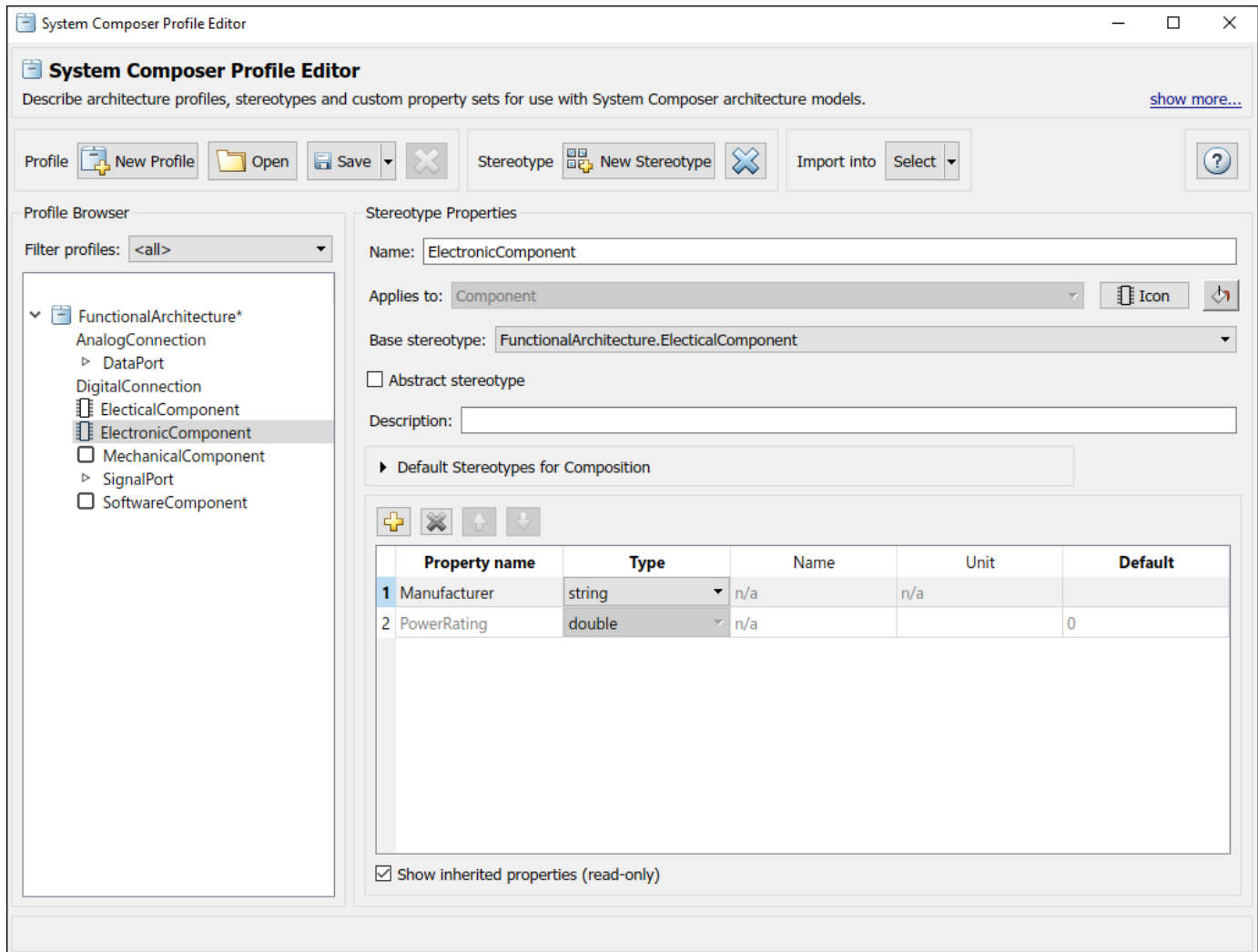
If a stereotype is no longer required for an element, remove it using the Property Inspector. Click **Select** next to the stereotype and choose **Remove**.



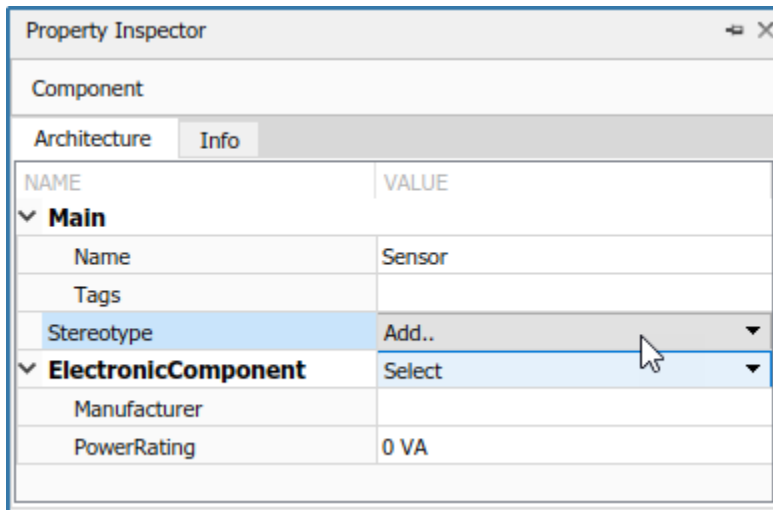
Extend Stereotypes

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.



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



See Also

editor | hasStereotype | hasProperty | systemcomposer.profile.Profile | systemcomposer.profile.Property | systemcomposer.profile.Stereotype

More About

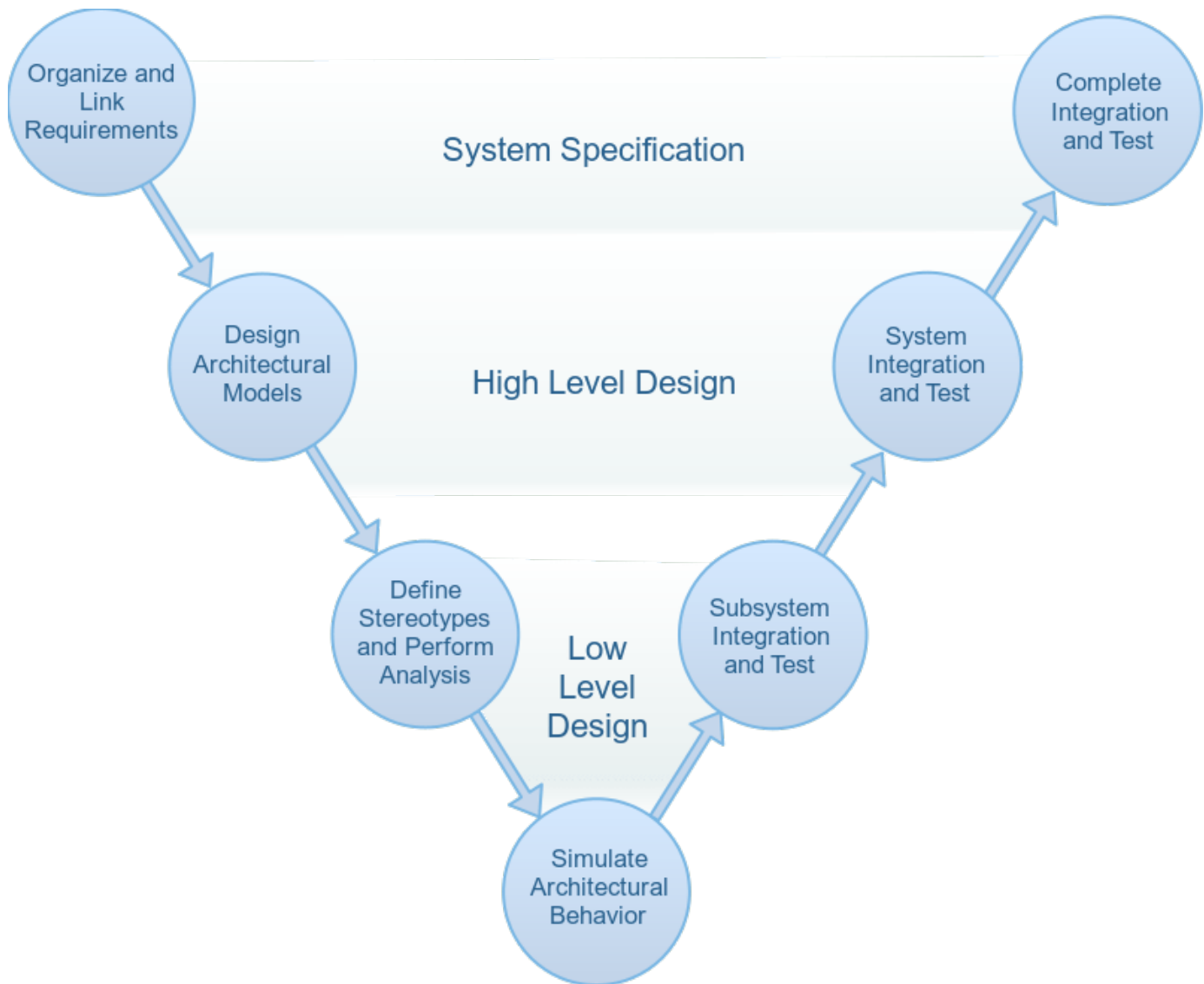
- “Define Profiles and Stereotypes” on page 4-2
- “Analyze Architecture” on page 7-2
- “Analysis Function Constructs” on page 7-9
- “Apply Stereotypes to Functions of Software Architectures” on page 8-29
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

Simulate Mobile Robot with System Composer Workflow

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®. Follow this tutorial for the early phase of development of an autonomous mobile robot.

- 1 “Organize and Link Requirements” on page 4-24: Set up the requirements based on market research using Requirements Toolbox™.
- 2 “Design Architecture Models” on page 4-27: Create architecture models to help organize algorithms and hardware.
- 3 “Define Stereotypes and Perform Analysis” on page 4-34: Define stereotypes and perform system analysis to ensure that the life expectancy of the durable components in the robot meets the customer-specified mean time before repair.
- 4 “Simulate Architectural Behavior” on page 4-43: Create a Simulink model to simulate realistic behavior of the mobile robot.

This workflow is represented by the left side of the model-based systems engineering (MBSE) design diagram.



See Also

More About

- "Model-Based Design with Simulink"
- "Organize System Composer Files in Projects" on page 10-2

Organize and Link Requirements

The first step in model-based systems engineering (MBSE) design using System Composer is to set up requirements. This functionality requires a Requirements Toolbox license.

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. This mobile robot example has three sets of requirements.

- 1 *Stakeholder needs*—A set of end-user needs. Stakeholders are interested in attributes of the mobile robot associated with endurance, payload, speed, autonomy, and reliability.
- 2 *System requirements*—A set of requirements that are linked closely with system-level design. System requirements include the derived requirements that describe how the system responds to stakeholder needs.
- 3 *Implementation requirements*—A set of requirements that specify subsystems in the model. Implementation requirements include specifications for the battery, structure, propulsion, path generation, position, controller, and component life for individual subsystems.

By linking one requirement set to another, each high-level requirement can be traced to implementation. As the MBSE design evolves, you can use iterative requirements analysis to enhance requirement traceability and coverage. You can use the traceability diagram to visualize requirement traceability. See “Visualize Links with a Traceability Diagram” (Requirements Toolbox).

Note This example uses Simscape™ blocks. If you do not have a Simscape license, you can open and simulate the model but can only make basic changes, such as modifying block parameters.

Link Stakeholder Requirements to System Requirements

The mobile robot example includes a functional, logical, and physical architecture that fulfill stakeholder needs, system requirements, and implementation requirements.

Load these systems in memory to view their requirement links:

- Functional architecture model
- Logical architecture model
- Physical architecture model

```
systemcomposer.loadModel("RobotFunctionalArchitecture");  
systemcomposer.loadModel("scMobileRobotLogicalArchitecture_SS");  
systemcomposer.loadModel("scMobileRobotHardwareArchitecture");
```

Load these requirement sets into memory:

- Stakeholder needs
- System requirements
- Implementation requirements

```
slreq.load("scMobileRobotStakeholderNeeds");  
slreq.load("scMobileRobotRequirements");  
slreq.load("scMobileRobotSubsystemRequirements");
```


Open the Requirements Editor (Requirements Toolbox).

slreq.editor

You can link stakeholder needs to derived requirements to keep track of high-level goals. The Mean Time Before Repair (MTBR) requirement, STAKEHOLDER-07, is refined by the Battery Life requirement, SYSTEM-REQ-09.

The screenshot displays the Requirements Editor interface. On the left, a table lists requirements under three categories: scMobileRobotRequirements, scMobileRobotStakeholderNeeds, and scMobileRobotSubsystemRequirements. The requirement STAKEHOLDER-07 (MTBR) is highlighted in blue. On the right, the detailed view for STAKEHOLDER-07 is shown, including its properties (Type: Functional, Index: 5.1, Custom ID: STAKEHOLDER-07, Summary: MTBR) and a description: "The robot should have a Mean Time Before Repair of 2 years." The Links section shows that STAKEHOLDER-07 is implemented by RobotFunctionalArchitecture and related to SYSTEM-REQ-09 Battery Life and SYSTEM-REQ-10 Sensor Life. A green arrow points to the link to SYSTEM-REQ-09 Battery Life.

Index	ID	Summary
scMobileRobotRequirements		
1	-	Endurance
2	-	Payload and Speed
3	-	Autonomy
4	-	Life Expectancy
scMobileRobotStakeholderNeeds		
1	STAKEHOLDER-01	Endurance
2	STAKEHOLDER-02	Payload
3	STAKEHOLDER-03	Operating Speed
4	-	Autonomy
	STAKEHOLDER-04	Transportation
	STAKEHOLDER-05	Autonomous Charging
	STAKEHOLDER-06	Collision Avoidance
5	-	Reliability
	STAKEHOLDER-07	MTBR
	STAKEHOLDER-08	MTBF
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:

- RobotFunctionalArchitecture

Related to:

- SYSTEM-REQ-09 Battery Life
- SYSTEM-REQ-10 Sensor Life

You can set a specific link type. To change link types, in the Requirements Editor (Requirements Toolbox), select **Show Links**. For more information, see "Requirement Links" (Requirements Toolbox).

4 Define Architectural Properties

The screenshot displays the Requirements Editor interface. On the left, a table lists various links between requirements and architectural elements. The table has columns for Label, Source, Type, and Destination. The right pane shows the 'Link' details for a selected link, including a 'Properties' section with a dropdown menu for 'Type'.

Label	Source	Type	Destination
scMobileRobotLogicalArchitecture_5G.slmx	Changed source: 0/6		Changed destination: 0/6
link #1	Motor Controller	Impements	SYSTEM-REQ-04 Nominal Operating Speed with Payload
link #4	Trajectory Follower	Impements	SYSTEM-REQ-08 Collision Avoidance
link #5	Trajectory Follower	Impements	SYSTEM-REQ-07 Path Following
link #7	Trajectory Generator	Impements	SYSTEM-REQ-05 Self-Localization
link #3	Trajectory Generator	Impements	SYSTEM-REQ-09 Charging Station Location
link #6	Trajectory Generator	Impements	SYSTEM-REQ-06 Path Generation
scMobileRobotHardwareArchitecture.slmx	Changed source: 0/8		Changed destination: 0/8
link #5	Battery	Impements	SYSTEM-REQ-01 Battery Capacity
link #9	Battery	Impements	SYSTEM-REQ-09 Battery Life
link #6	Charge Board	Impements	SYSTEM-REQ-02 Battery Charge Time
link #3	Lidar Sensor	Impements	SYSTEM-REQ-10 Sensor Life
link #7	Payload	Impements	SYSTEM-REQ-03 Maximum Payload
link #2	RGB Camera	Impements	SYSTEM-REQ-10 Sensor Life
link #17	scMobileRobotHardwareArchitecture	Impements	SYSTEM-REQ-11 Mechanical Component Life
link #8	Wheels	Impements	SYSTEM-REQ-04 Nominal Operating Speed with Payload
RobotFunctionalArchitecture.slmx	Changed source: 0/11		Changed destination: 0/11
link #7	Actuate Motors	Impements	STAKEHOLDER-03 Operating Speed
link #10	Check Safety	Impements	STAKEHOLDER-06 Collision Avoidance
link #14	Compute Self Position	Impements	STAKEHOLDER-04 Transportation
link #8	Drive	Impements	STAKEHOLDER-03 Operating Speed
link #12	Identify Target Position	Impements	STAKEHOLDER-05 Autonomous Charging
link #13	Identify Target Position	Impements	STAKEHOLDER-04 Transportation
link #11	Plan Path	Impements	STAKEHOLDER-04 Transportation
link #5	RobotFunctionalArchitecture	Impements	STAKEHOLDER-01 Endurance
link #6	RobotFunctionalArchitecture	Impements	STAKEHOLDER-02 Payload
link #15	RobotFunctionalArchitecture	Impements	STAKEHOLDER-07 NTBR
link #16	RobotFunctionalArchitecture	Impements	STAKEHOLDER-08 NTBF
scMobileRobotStakeholderNeeds.slmx	Changed source: 0/19		Changed destination: 0/19
link #7	STAKEHOLDER-01 Endurance	Related to	SYSTEM-REQ-01 Battery Capacity
link #3	STAKEHOLDER-01 Endurance	Related to	SYSTEM-REQ-02 Battery Charge Time
link #4	STAKEHOLDER-02 Payload	Related to	SYSTEM-REQ-03 Maximum Payload
link #6	STAKEHOLDER-02 Payload	Related to	SYSTEM-REQ-13 Motor Type
link #8	STAKEHOLDER-03 Operating Speed	Related to	SYSTEM-REQ-04 Nominal Operating Speed with Payload

The right pane shows the 'Link' details for the selected link. The 'Type' dropdown menu is open, showing options: Confirmed By, Derives, Impements, Refines, Related to, and Verifies. The 'Destination' field is set to '04 Transportation'.

To return to interacting with requirements, in the Requirements Editor (Requirements Toolbox), select **Show Requirements**. The Transportation stakeholder needs requirement, STAKEHOLDER-04, will be implemented by the Localization 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 STAKEHOLDER-04 and select **Create a link from SYSTEM-REQ-05 to STAKEHOLDER-04**.

See Also

slreq.editor | slreq.load | systemcomposer.loadModel

More About

- “Manage Requirements” on page 2-8
- “Link and Trace Requirements” on page 2-2
- “Design Insulin Infusion Pump Using Model-Based Systems Engineering” on page 7-23

Design Architecture Models

Architecture models in System Composer describe a system at different levels of abstraction. This mobile robot example presents three architectures:

- 1 *Functional architecture* describes high-level functions and the relationships between them.
- 2 *Logical architecture* describes data exchange between electronic hardware and software components in each subsystem.
- 3 *Physical architecture* describes the physical hardware or platform needed for the robot.

Note This example uses Simscape blocks. If you do not have a Simscape license, you can open and simulate the model but can only make basic changes, such as modifying block parameters.

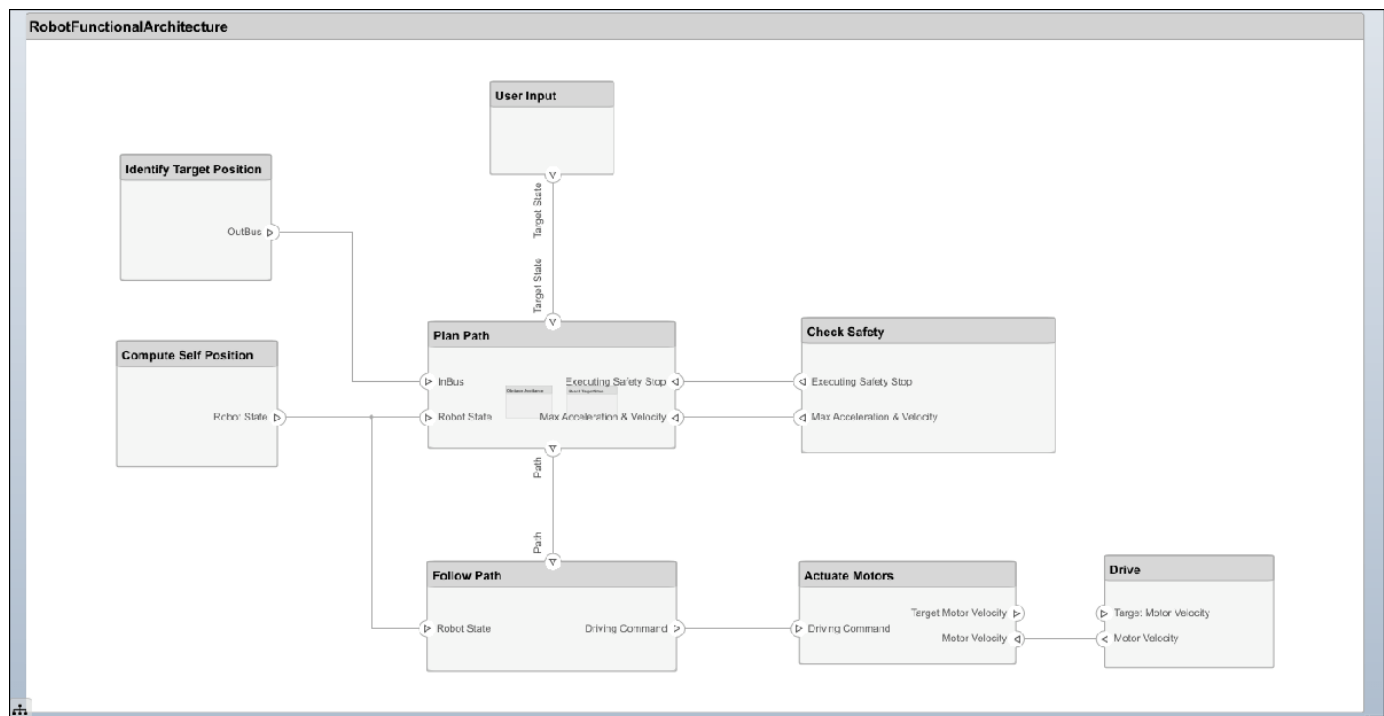
Design, Specify, and Allocate Architecture Models

The mobile robot example includes a functional, logical, and physical architecture with requirements linked to components and model-to-model allocations defined.

Functional Architecture Model for Mobile Robot

The functional architecture model describes functional dependencies: controlling a mobile robot autonomously, localization, path-planning, and path-following. To open the functional architecture model, double-click the file or run this command.

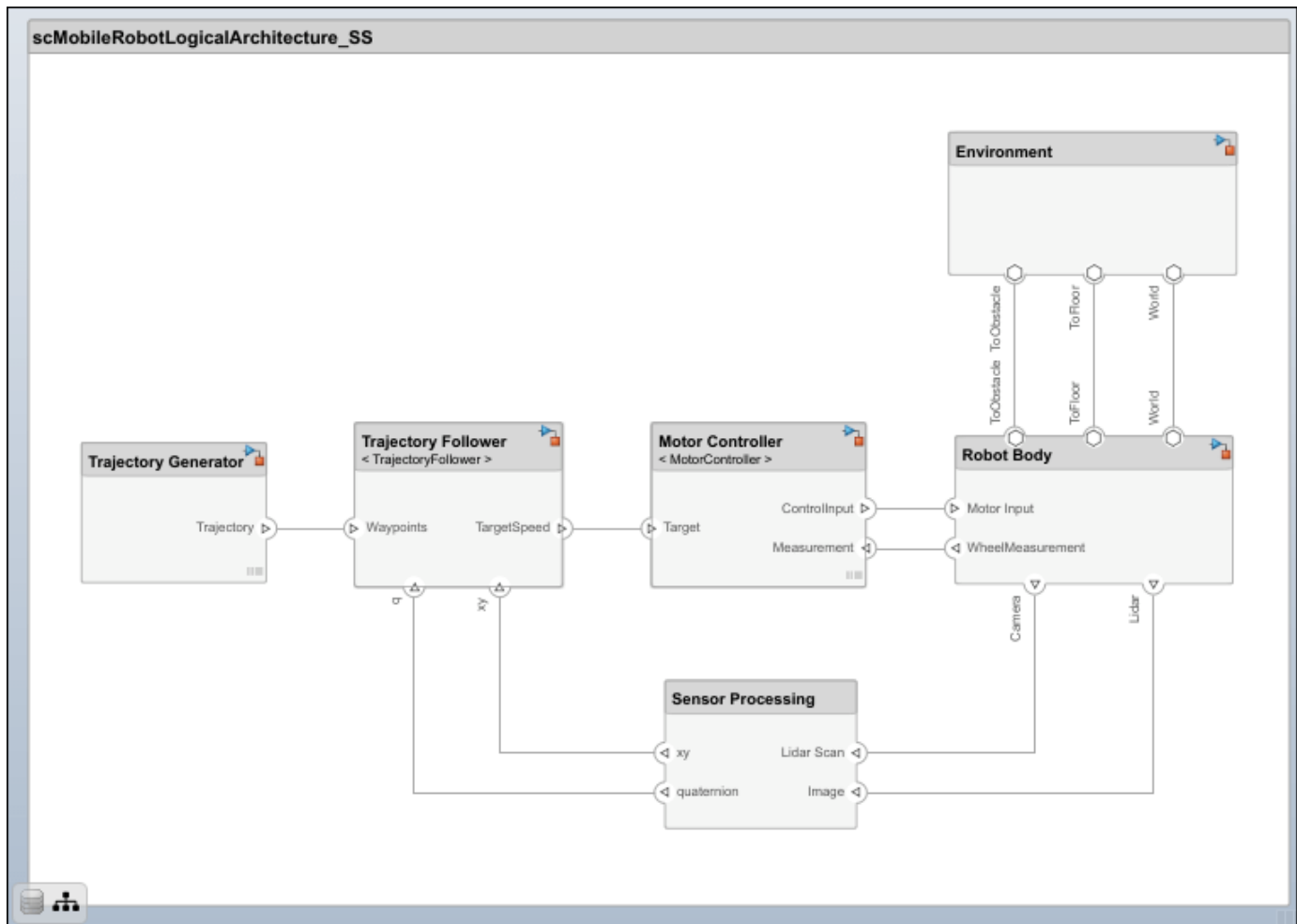
```
systemcomposer.openModel("RobotFunctionalArchitecture");
```



Logical Architecture Model for Mobile Robot

The logical architecture model describes the behavior of the mobile robot system for simulation: trajectory generator, trajectory follower, motor controller, sensor algorithm, and robot and environment. The connections represent the interactions in the system. To open the logical architecture model, double-click the file or run this command.

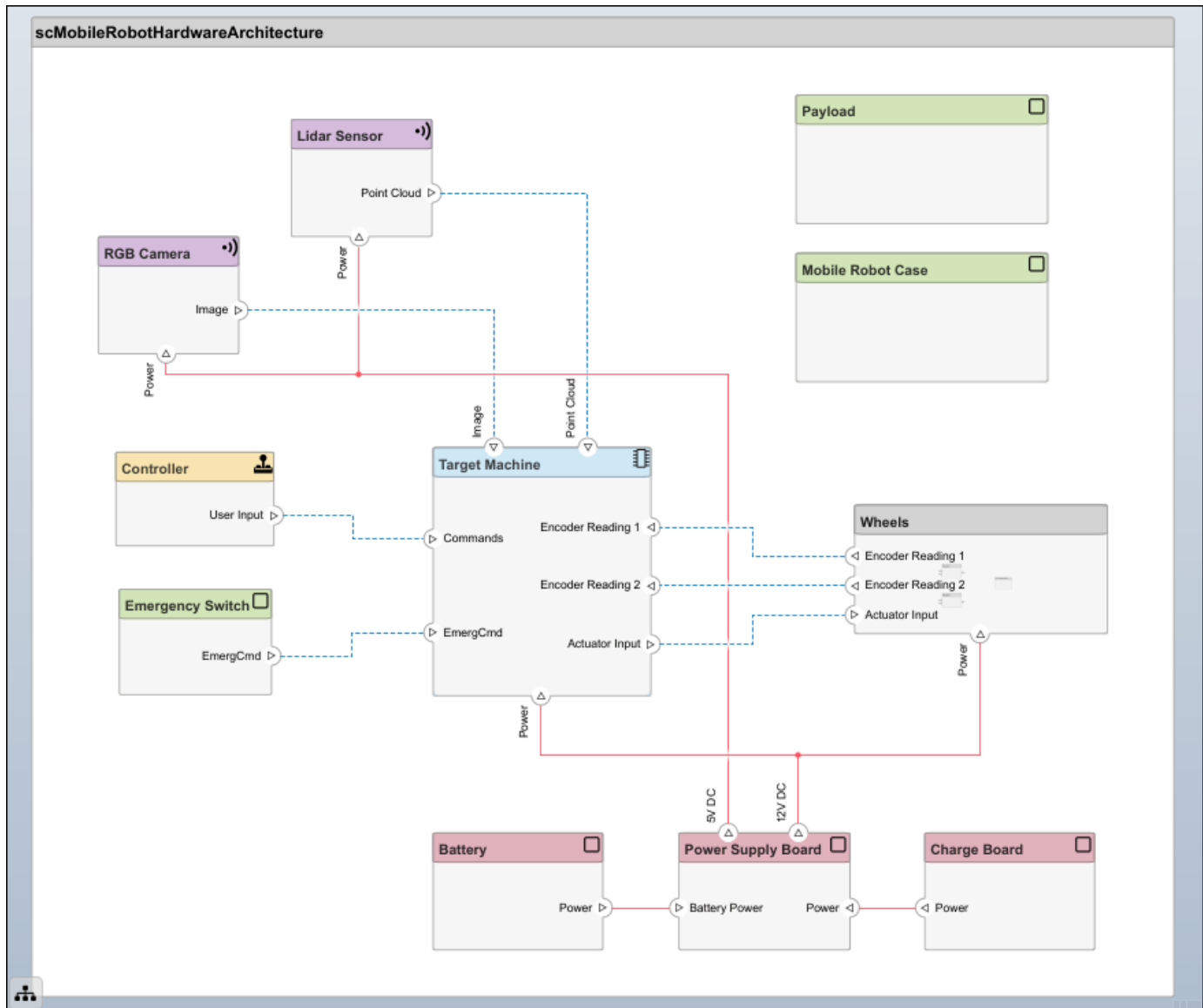
```
systemcomposer.openModel("scMobileRobotLogicalArchitecture_SS");
```



Physical Architecture Model for Mobile Robot

The physical architecture model describes the hardware components and their connections: the sensor, actuators, and embedded processor. The colors and icons indicate the stereotypes used for each element. To open the physical architecture model, double-click the file or run this command.

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



Link Requirements to Components

Requirement traceability involves linking technical requirements to components and ports in architecture models, thereby allowing the connection between an early requirements phase and system-level design. You can easily track whether a requirement is met by connecting components back to stakeholder needs. You can add requirement links by dragging requirements to a component.

To view requirements, open the Requirements Manager by navigating to **Apps > Requirements Manager**.

The **Identify Target Position** component in the functional architecture model implements the **Autonomous Charging** requirement, **STAKEHOLDER-05**. To show or hide linked requirements, click the requirement icon on the top-right corner of a component.

The screenshot shows a software development tool interface. At the top, there are tabs for 'RobotFunctionalArchitecture' and 'Plan Path'. The main workspace displays a requirement diagram. A requirement box titled 'Identify Target Position' is connected to a stakeholder box titled 'STAKEHOLDER-05: Autonomous ...' via an 'OutBus' and an 'IMPLEMENTS' relationship. A pink arrow points from the stakeholder box to the requirement box. To the right of the requirement box, there are two links: 'STAKEHOLDER-05 Autonomous Charging' and 'STAKEHOLDER-04 Transportation', each with a 'Hide' button. Below the workspace, there is a 'Requirements - RobotFunctionalArchitecture' section with a 'View: Requirements' dropdown and a toolbar. At the bottom, there is a table with columns for Index, ID, and Summary.

Index	ID	Summary
1	STAKEHOLDER-01	Endurance
2	STAKEHOLDER-02	Payload
3	STAKEHOLDER-03	Operating Speed
4	-	Autonomy
4.1	STAKEHOLDER-04	Transportation
4.2	STAKEHOLDER-05	Autonomous Charging
4.3	STAKEHOLDER-06	Collision Avoidance
5	-	Reliability

You can view the requirements linked to the hardware architecture model in the Requirements Browser. After selecting STAKEHOLDER-04, only components related to the Transportation requirement are shown.

The screenshot displays the RobotFunctionalArchitecture software interface. The main workspace shows a functional architecture diagram with components and their relationships. The 'Identify Target Position' component is connected to 'STAKEHOLDER-05: Autonomous ...' via an 'IMPLEMENTS' relationship. The 'Compute Self Position' component is connected to 'STAKEHOLDER-04: Transportation' via an 'IMPLEMENTS' relationship. The 'Plan Path' component is connected to 'STAKEHOLDER-04: Transportation' via an 'IMPLEMENTS' relationship. The 'Plan Path' component also has inputs for 'InBus', 'Robot State', and 'Path', and outputs for 'Executing Safety Stop', 'Max Acceleration & Velocity', and 'Path'. The 'User Input' component is connected to 'Plan Path' via a 'Target State' relationship. The 'Robot State' component is connected to 'Plan Path' via a 'Target State' relationship. The 'OutBus' component is connected to 'Identify Target Position' via an 'IMPLEMENTS' relationship. The 'Requirements - RobotFunctionalArchitecture' browser at the bottom shows a list of requirements, with STAKEHOLDER-04 (Transportation) selected. The Properties Inspector on the right shows details for STAKEHOLDER-04, including its index (4.1), custom ID (STAKEHOLDER-04), and summary (Transportation). The Description tab shows the rationale: 'The robot must be able to relocate an object in the warehouse from one positions to another with a tolerance of ±0.1 m.' The Links section shows that STAKEHOLDER-04 is implemented by 'Plan Path', 'Identify Target Position', and 'Compute Self Position'. It is also related to 'SYSTEM-REQ-05 Self-Localizati...'.

Index	ID	Summary
2	STAKEHOLDER-02	Payload
3	STAKEHOLDER-03	Operating Speed
4	-	Autonomy
4.1	STAKEHOLDER-04	Transportation
4.2	STAKEHOLDER-05	Autonomous Charging
4.3	STAKEHOLDER-06	Collision Avoidance
5	-	Reliability
5.1	STAKEHOLDER-07	MTBR
5.2	STAKEHOLDER-08	MTBF

Allocate Architectures

You can allocate functional components to physical components using model-to-model allocations. To open the Allocation Editor, navigate to **Modeling > Allocation Editor**, or run this command.

```
systemcomposer.allocation.editor
```

Load the allocation sets.

```
allocSetFunc = systemcomposer.allocation.load("FunctionalAllocation");
allocSetPhys = systemcomposer.allocation.load("PhysicalAllocation");
```

Allocate Functional to Physical Architectures

Click Scenario 1 under the FunctionalAllocation allocation set.

Select the **Component** in the **Row Filter** and **Column Filter** sections. The Allocation Editor tool allows you to link components between different architecture models to establish traceability for your project. Double-click the boxes in the allocation matrix to allocate or deallocate two elements.

	<input type="checkbox"/> scMobileRobotLogicalArchitecture_SS <input type="checkbox"/> Robot Body <input type="checkbox"/> Sensor Processing <input type="checkbox"/> Trajectory Generator			
▼ <input type="checkbox"/> RobotFunctionalArchitecture				
▼ <input type="checkbox"/> Identify Target Position				⬆
▼ <input type="checkbox"/> Actuate Motors				
▼ <input type="checkbox"/> User Input				⬆
▼ <input type="checkbox"/> Follow Path				
▼ <input type="checkbox"/> Plan Path			⬆	
<input type="checkbox"/> Reach Target State			⬆	
<input type="checkbox"/> Obstacle Avoidance			⬆	
▼ <input type="checkbox"/> Compute Self Position				⬆
▼ <input type="checkbox"/> Check Safety				⬆
▼ <input type="checkbox"/> Drive		⬆		

In the functional architecture, the trajectory generator requires components such as Identify Target Position, User Input, and Compute Self Position, so these components are allocated to the Trajectory Generator component in the logical architecture.

Allocate Logical to Physical Architectures

Click Scenario 1 under the PhysicalAllocation allocation set.

	scMobileRobotHardwareArchitecture	Controller	Lidar Sensor	Target Machine	RGB Camera	Mobile Robot Case
scMobileRobotLogicalArchitecture_SS						
Adapter						
Motor Controller						
Robot Body				↑		↑
Trajectory Follower						
Environment						
Sensor Processing						
Scan Matching Algorithm			↑			
Sensor Fusion				↑		
Alignment Algorithm					↑	
Trajectory Generator		↑		↑		

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 Robot Body, Sensor Fusion, and Trajectory Generator are allocated to the Target Machine component in the physical architecture model.

See Also

allocate | addComponent | addPort | connect

More About

- “Compose Architecture Visually” on page 1-2
- “Link and Trace Requirements” on page 2-2
- “Create and Manage Allocations” on page 6-2

Define Stereotypes and Perform Analysis

Stereotypes add an additional layer of metadata to components, ports, and connectors in System Composer. A stereotype is a custom extension of the modeling language. Stereotypes provide a mechanism to extend the architectural language elements by adding domain-specific metadata. The hardware architecture model provides a basis to understand the stereotypes applied to model elements, create filtered views based on the stereotypes, and perform an analysis on the model.

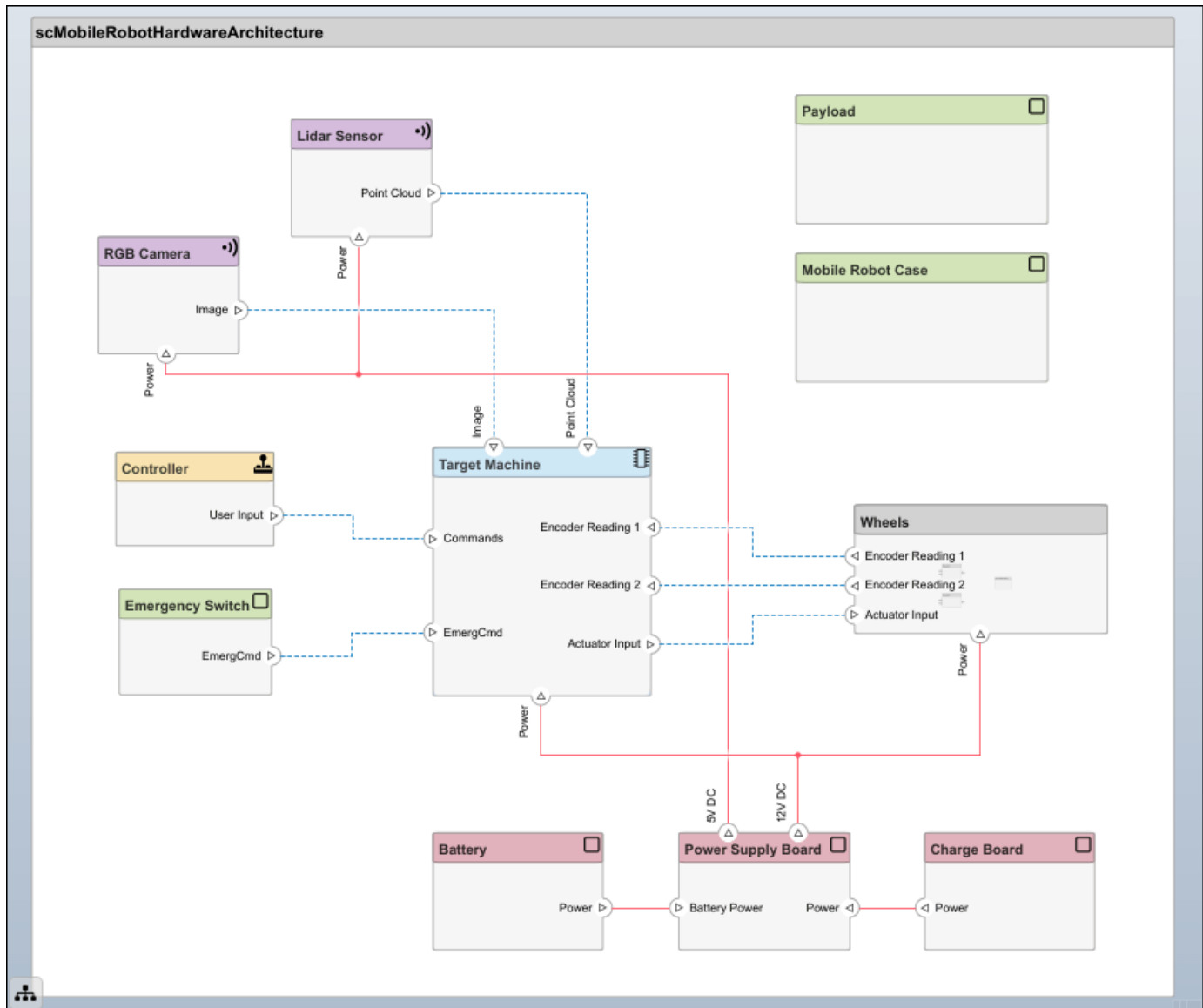
Define Stereotypes and Perform Analysis

This mobile robot example includes a profile applied to a physical architecture with stereotypes and properties defined. You can use views to display stakeholder concerns. Perform a remaining useful life (RUL) analysis on the life expectancy of the hardware components.

Hardware Architecture Model for Mobile Robot

The hardware architecture model describes the hardware components and their connections: the sensor, actuators, and embedded processor. The colors and icons indicate the stereotypes used for each element. To open the hardware architecture model, double-click the file or run this command.

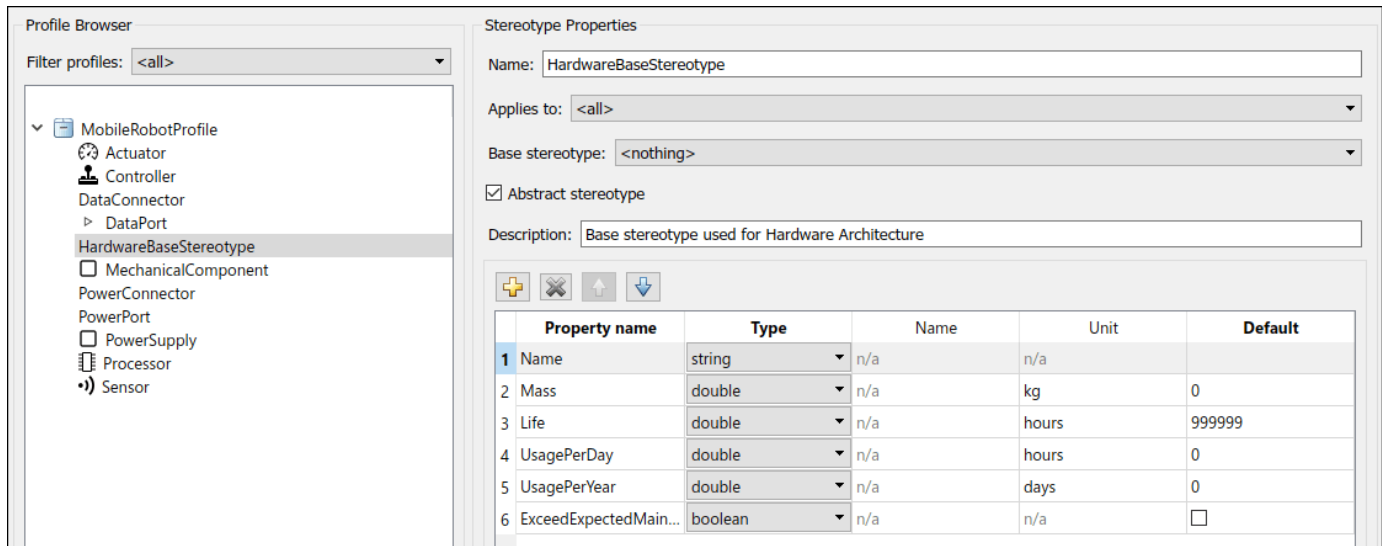
```
systemcomposer.openModel("scMobileRobotHardwareArchitecture");
```



View Stereotypes and Properties in Profile Editor

In this example, the `HardwareBaseStereotype` stereotype is defined as an abstract stereotype and is extended to connector and component stereotypes. For example, a `DataConnector` stereotype is a connector stereotype that inherits the `HardwareBaseStereotype`.

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, navigate to **Modeling > Profile Editor**.

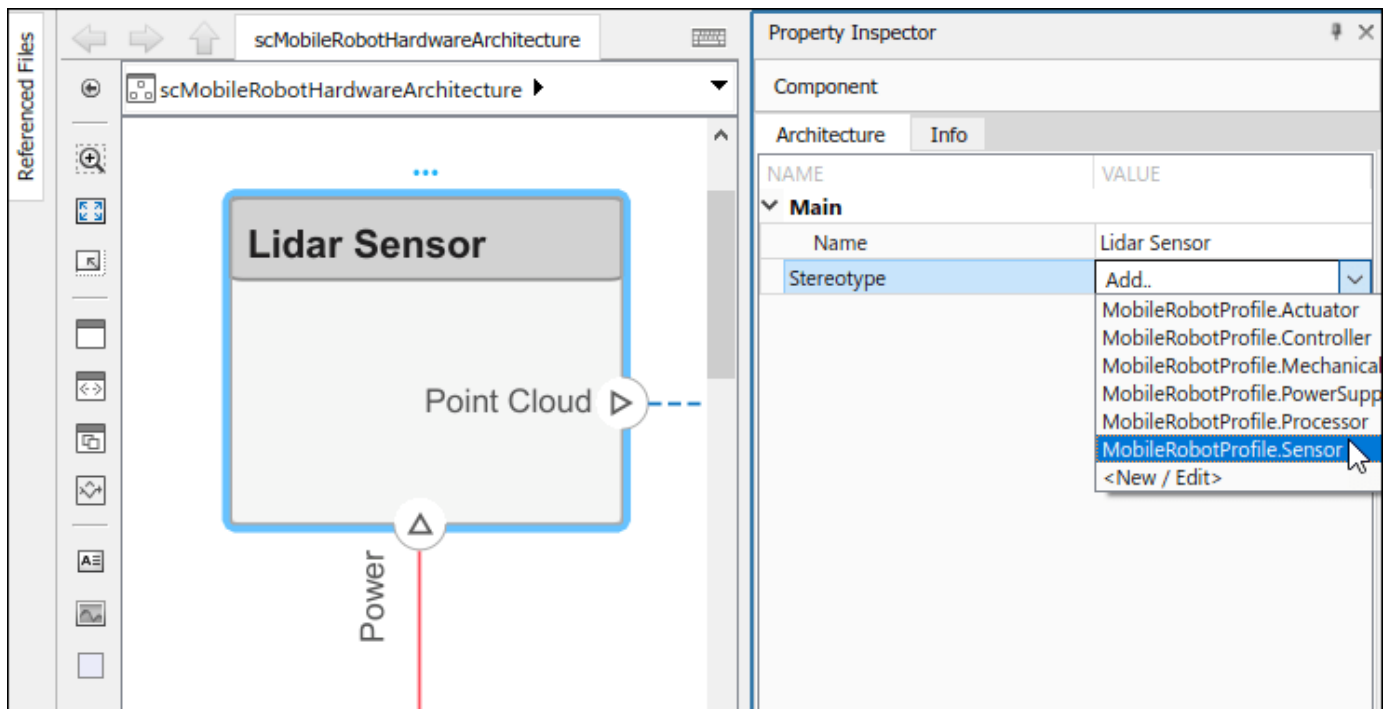


In addition to properties like name and mass, the `DataConnector` stereotype has a property of enumeration type, `TypeOfConnection`, that describes which of the three connection types it uses: RS232, Ethernet, or USB. To generate custom data types, create a script similar to `ConnectorType.m`. For more information, see “Simulink Enumerations”.

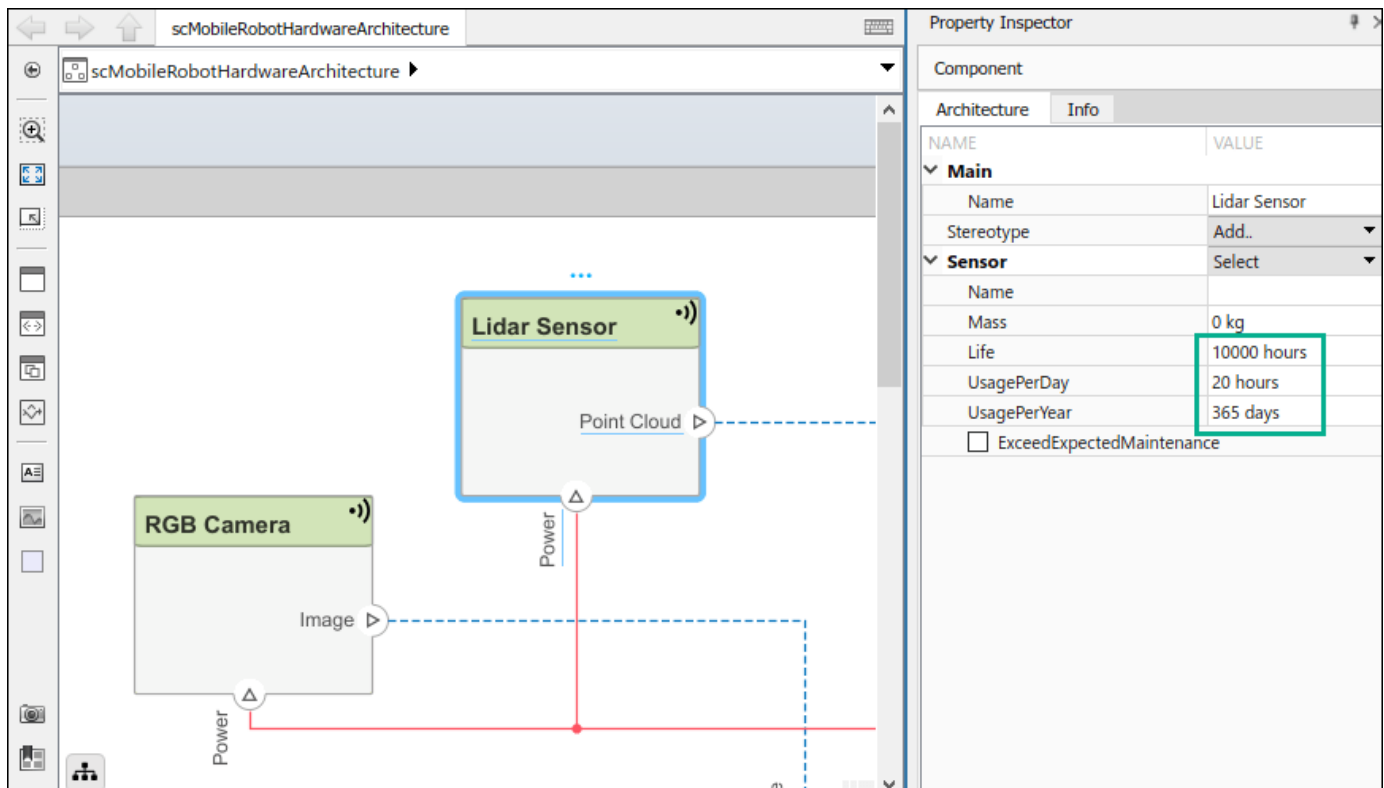
Apply Stereotypes to Elements in Model

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

To add stereotypes to elements, select the element in the diagram. In the Property Inspector, select **Main > Stereotype**. You can apply multiple stereotypes to the same element. Apply the `MobileRobotProfile.Sensor` stereotype to the Lidar Sensor component to add properties.




Some components remain 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.

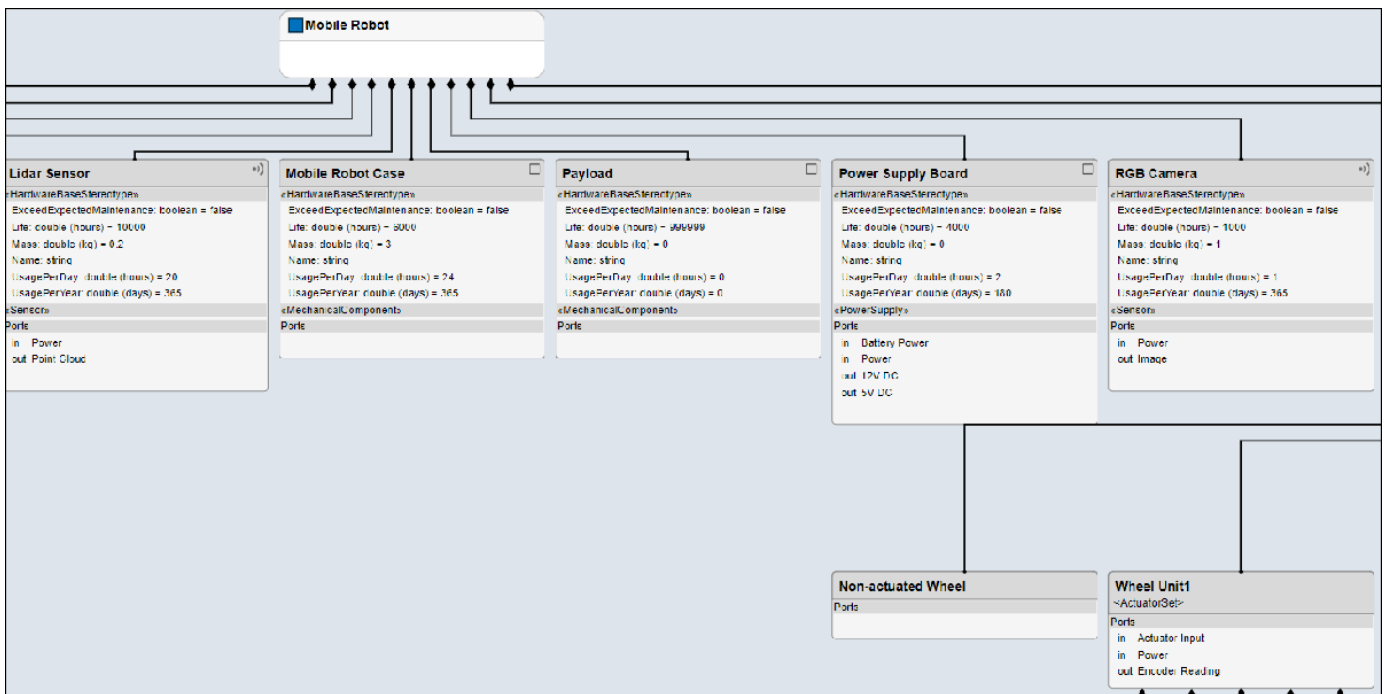


The property `ExceedExpectedMaintenance` is set to false by default. This property will update when you run your analysis.

Architecture Views for Hardware Architecture Model

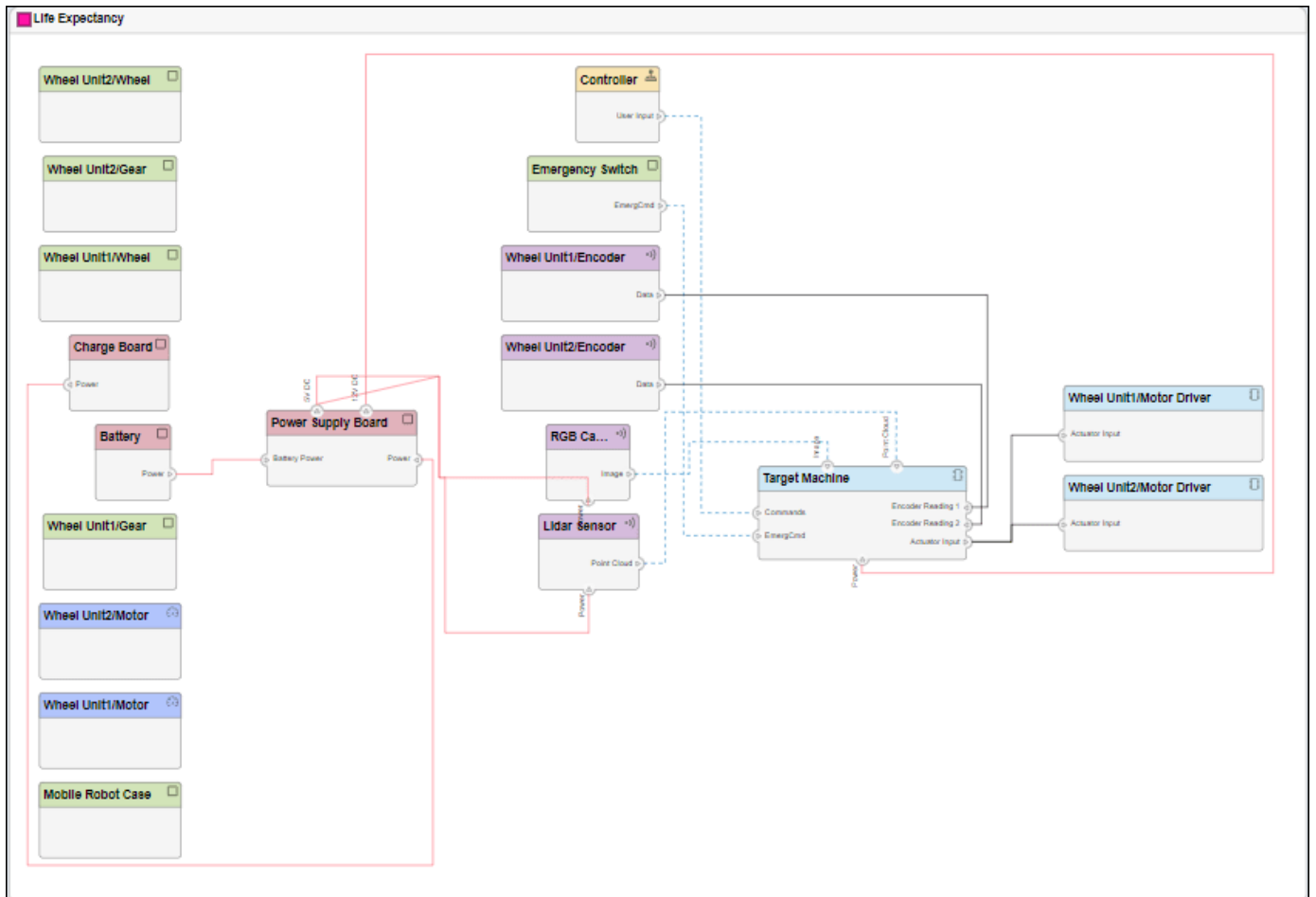
Use the Architecture Views Gallery to review changes you make in the architecture model. Architecture views allow you to create filtered views and thereby focus on few elements of the model, which enables you to navigate a complex model more easily.

- 1 To open the Architecture Views Gallery, navigate to **Modeling > Architecture Views**.
- 2 Select **New > View** to create a new view.
- 3 Name the view in the **View Properties** pane on the right.
- 4 In the bottom pane, under **View Configurations > Filter**, select from the list **Add Component Filter > Select All Components** to show all components in the view. Select **Apply** .
- 5 Select the **Component Hierarchy** view. The hierarchy of the components is flattened to show all subcomponents in one view.



- 7 You can apply a filter to view components with the Life Expectancy requirement. Select **New > View** and name the view in the **View Properties** pane on the right.
- 8 In the bottom pane under **View Configurations > Filter**, select **Add Component Filter**.

- 10 Select **Apply** . Observe the components with the Life property defined.




The components with the Life property defined are components for which expected time before first maintenance is a concern.

Analyze Hardware Components for Life Expectancy

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 two years in the analysis function. Navigate to **Modeling > Analysis Model** to open the Instantiate Architecture Model tool.

Select all stereotypes to make them available on the instance model. Select `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 iteration order **Pre-order**.

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
 - DataPort
 - HardwareBaseStereotype
 - MechanicalComponent
 - PowerConnector
 - PowerPort
 - PowerSupply
 - Processor
 - Sensor

Strict Mode

Don't see your profile? [Profile Editor ...](#)


Step 2: Configure Analysis

Function

Analysis function:
   

Function arguments (comma-separated):



Model Iteration

Iteration Order: 

Instance Model Properties

Name:

Normalize Units

 Cancel  Instantiate

Click **Instantiate** to instantiate the model and open the Analysis Viewer tool. 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 two years. Click **Analyze** to perform the calculation.

Instances	TypeOfConnection	ExceedExpectedMaintenance	Life	Mass	UsagePerDay	UsagePerYear
scMobileRobotHardwareArchitecture						
Battery		<input type="checkbox"/>	10000	0	24	365
Charge Board		<input checked="" type="checkbox"/>	2000	0	2	180
Controller		<input checked="" type="checkbox"/>	1000	0	1	365
Emergency Switch		<input checked="" type="checkbox"/>	3000	0	0.5	52
Lidar Sensor		<input type="checkbox"/>	10000	0.2	20	365
Mobile Robot Case		<input type="checkbox"/>	6000	3	24	365
Payload		<input checked="" type="checkbox"/>	999999	0	0	0
Power Supply Board		<input checked="" type="checkbox"/>	4000	0	2	180
RGB Camera		<input checked="" type="checkbox"/>	1000	1	1	365
Target Machine		<input checked="" type="checkbox"/>	5000	2.2	0.5	180
Wheels						
Wheel Unit1						
Encoder		<input checked="" type="checkbox"/>	15000	0	20	365
Gear		<input type="checkbox"/>	5000	0	20	365
Motor		<input checked="" type="checkbox"/>	30000	0	20	365
Motor Driver		<input type="checkbox"/>	3000	0	20	365
Wheel		<input type="checkbox"/>	5000	0	20	365
Wheel Unit2						
Encoder		<input checked="" type="checkbox"/>	15000	0	20	365
Gear		<input type="checkbox"/>	5000	0	20	365
Motor		<input checked="" type="checkbox"/>	30000	0	20	365
Motor Driver		<input type="checkbox"/>	3000	0	20	365
Wheel		<input type="checkbox"/>	5000	0	20	365

The components for which usage is not defined are components that last significantly longer than the expected time and are therefore excluded from analysis. The analysis function calculates whether the time before first maintenance for each component and connector will exceed `Life`, which is set to two years. The unchecked boxes indicate that components and connectors will need maintenance within two years.

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 reverts back to its default value. Conversely, when you click **Update** the property values in the hardware architecture source update according to the instance model.

See Also

[applyProfile](#) | [applyStereotype](#) | [openViews](#) | [instantiate](#)

More About

- “Define Profiles and Stereotypes” on page 4-2
- “Create Architecture Views Interactively” on page 9-5
- “Analyze Architecture” on page 7-2
- “Analysis Function Constructs” on page 7-9

Simulate Architectural Behavior

To simulate the mobile robot logical architecture, link Simulink models to the components or add Simulink subsystem components. For details, see “Describe Component Behavior Using Simulink” on page 5-2. These models act as Simulink behaviors and can be simulated in System Composer by clicking **Run**. The simulation shows how well the mobile robot follows a trajectory created by a controller to avoid an obstacle.

Note This example uses Simscape blocks. If you do not have a Simscape license, you can open and simulate the model but can only make basic changes, such as modifying block parameters.

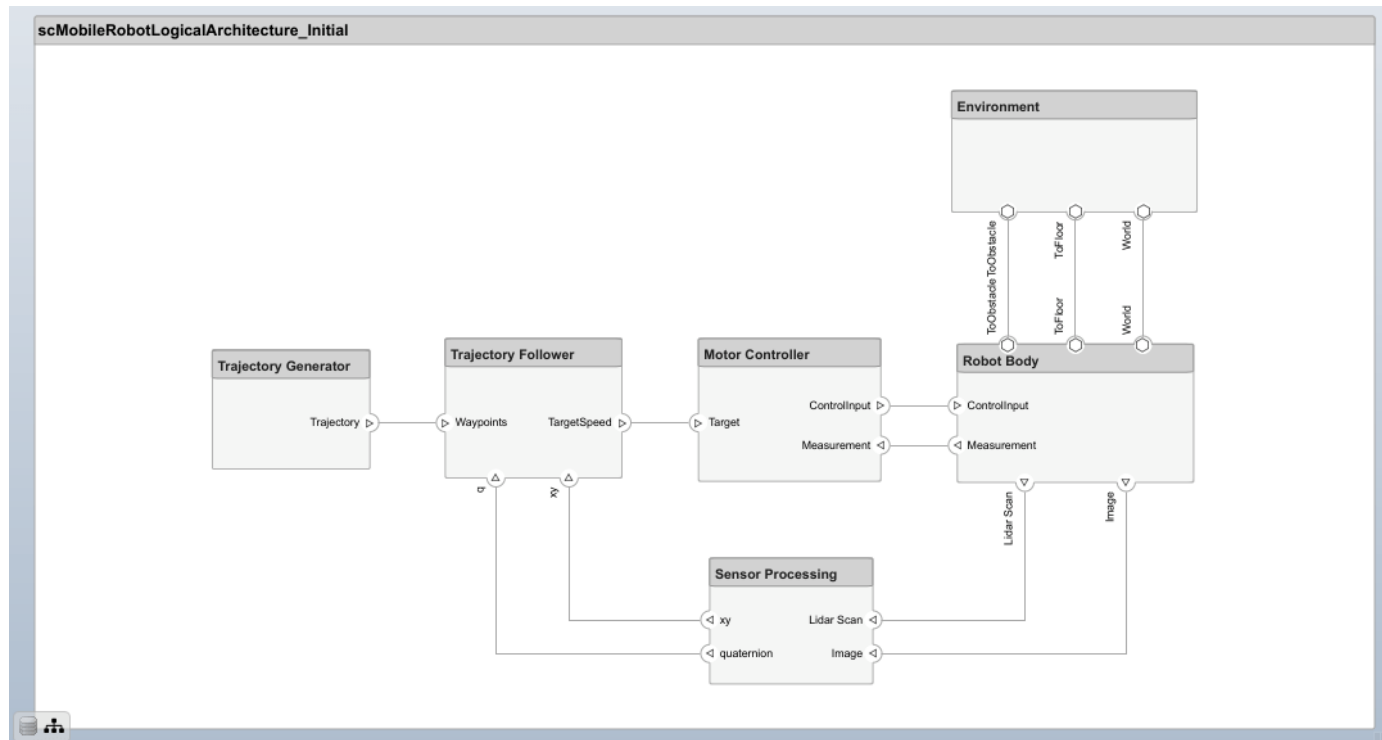
Simulate Architectural Behavior

The mobile robot example includes a logical architecture with component behaviors defined. You can run the simulation to watch the mobile robot avoid an obstacle.

Add Behavior to Logical Architecture

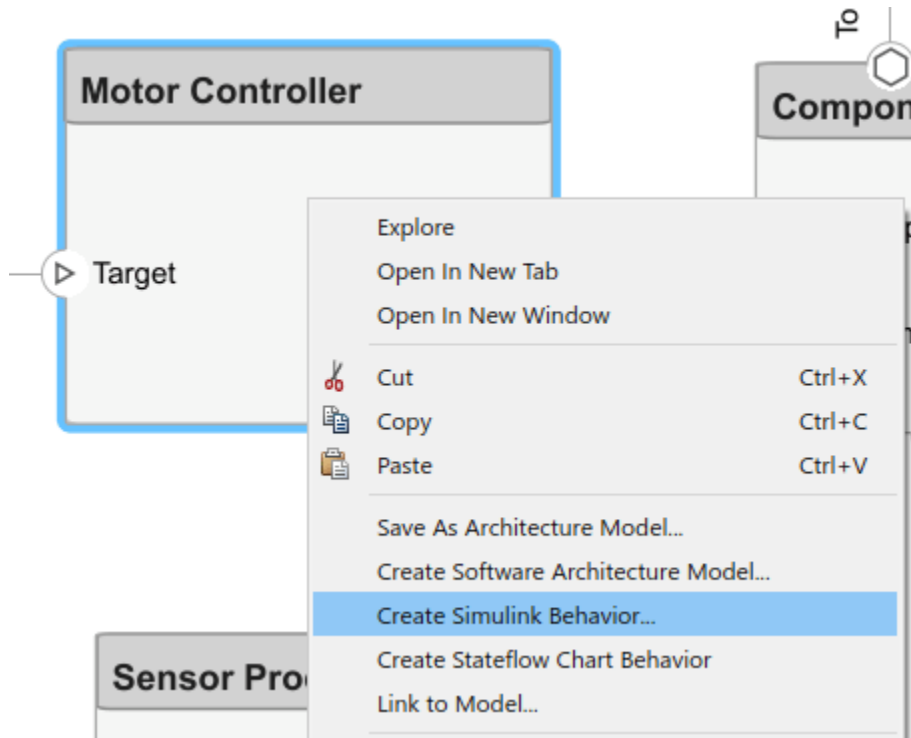
The logical architecture model describes the behavior of the mobile robot system for simulation: trajectory generator, trajectory follower, motor controller, sensor algorithm, and robot and environment. The connections represent the interactions in the system. Open the logical architecture model without any behaviors, double-click the file or run this command.

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



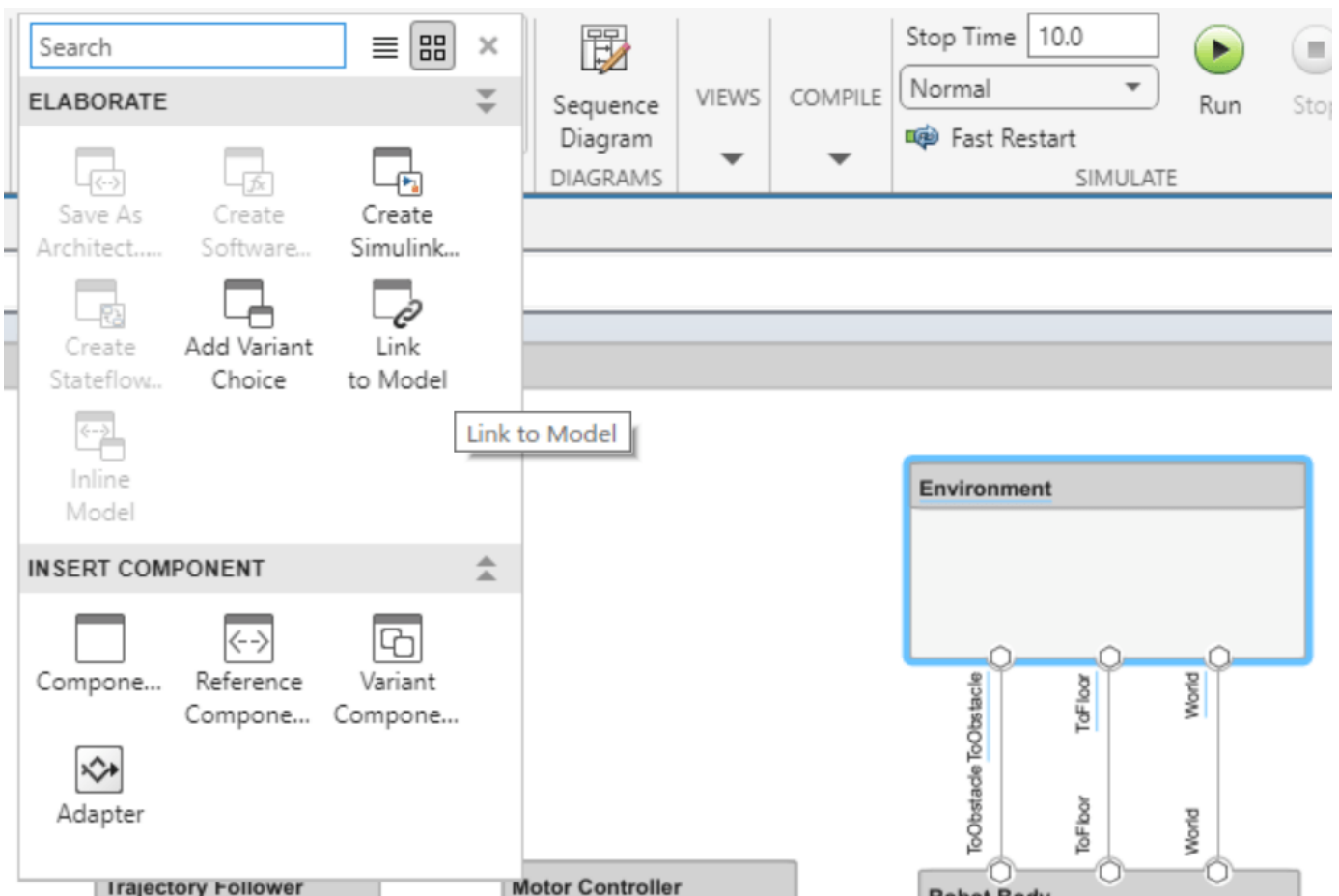
The architecture model describes the behavior of the robot, but no behavior is actually added to the architecture yet. By adding Simulink or Stateflow® behavior, the logical architecture can also be simulated.

Create a new behavior based on the interface of a component. If a model or subsystem file already exists for the behavior, use **Link To Model** to link to the existing model or subsystem. To create new subsystem reference behavior for the **Motor Controller** component, right-click and select **Create Simulink Behavior**, or, on the toolstrip, click **Create Simulink Behavior**. For more information, see “Describe Component Behavior Using Simulink” on page 5-2.



You can create Simulink behaviors using multiple methods: **Subsystem**, **Subsystem Reference**, and **Model Reference**. Use **Subsystem** to create a subsystem component behavior that is part of the parent architecture model. Use the **Subsystem Reference** or **Model Reference** option to save the behavior as a separate artifact and reuse the behavior. Physical ports can only cross subsystem boundaries, so for physical systems, **Subsystem Reference** or **Subsystem** are recommended.

If you already have a behavior defined in a model file or subsystem file, use **Link To Model** to link a component to the corresponding file. On the toolstrip, click **Link to Model**, or right-click the **Environment** component and select **Link to Model** to link to the **Environment.slx** subsystem file.



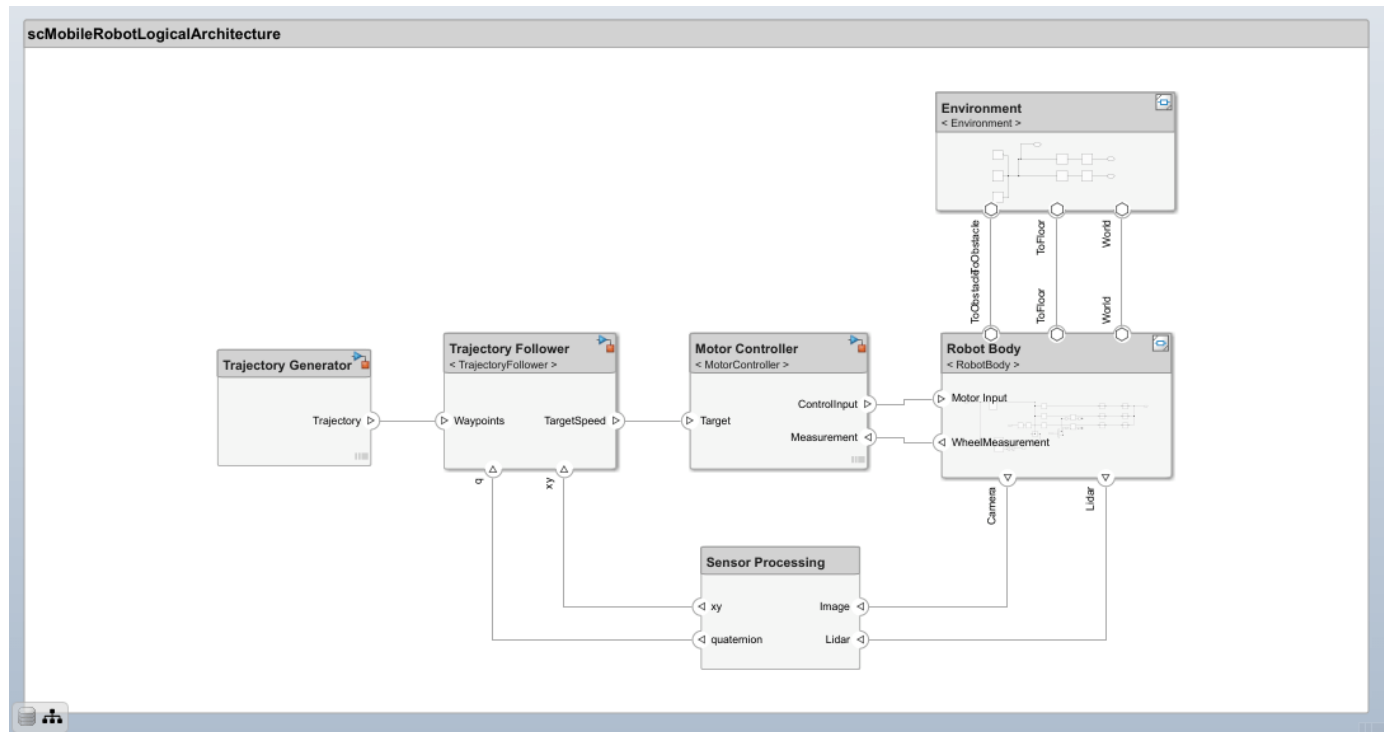
Logical Architecture Model for Mobile Robot

Open the final logical architecture where behavior is added to all the components.

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

The Robot Body, Trajectory Generator, and Environment components are Simulink subsystem components that support physical ports. The Trajectory Follower and Motion Controller components are represented as Simulink models linked to the components as referenced models.

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

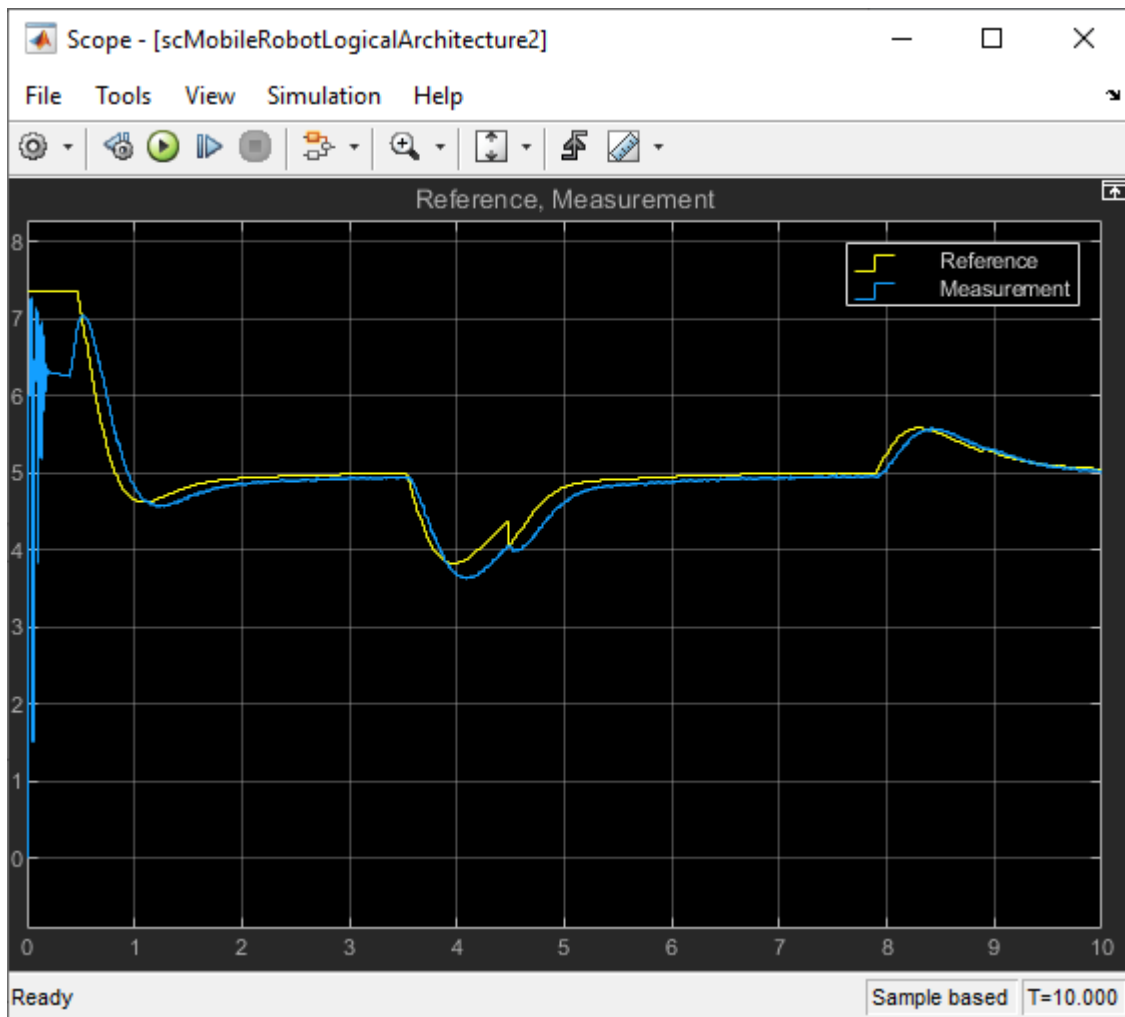



Run Simulation Using Logical Architecture

Once behavior models are linked, you can simulate the architecture model just like any other Simulink model by clicking **Run**. Simulation verifies requirements such as Transportation, Collision Avoidance, and Path Generation.

```
sim scMobileRobotLogicalArchitecture;
```

The scope from the MotorController component behavior shows how a simple P-gain controller performs to follow the reference velocity for one of the wheels on the robot.



In the Mechanics Explorer (Simscape Multibody), switch to the isometric view by selecting . Watch the mobile robot avoid an obstacle.



See Also

`createSimulinkBehavior`

More About

- “Define Port Interfaces Between Components” on page 3-2
- “Explore Simulink Bus Capabilities”
- “Describe Component Behavior Using Simulink” on page 5-2

Use Simulink Models with System Composer

- “Describe Component Behavior Using Simulink” on page 5-2
- “Use Parameters to Store Instance Values with Components” on page 5-13
- “Extract Architecture of Simulink Model Using System Composer” on page 5-17
- “Describe Component Behavior Using Stateflow Charts” on page 5-23
- “Extract Architecture from Simulink Model” on page 5-28
- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Author Sequence Diagram for Traffic Light Example” on page 5-38
- “Use Sequence Diagrams with Architecture Models” on page 5-51
- “Implement Sequence Diagram Fragments” on page 5-65
- “Describe Component Behavior Using Simscape” on page 5-75
- “Synchronize Sequence Diagrams and Architecture Models” on page 5-81

Describe Component Behavior Using Simulink

System design and architecture definitions can involve a behavior definition for some components, such as the algorithm for a data processing component. Define components in System Composer architecture models as behaviors using Simulink subsystem components that are part of the parent model, or referenced behaviors by linking components to Simulink models or subsystems.

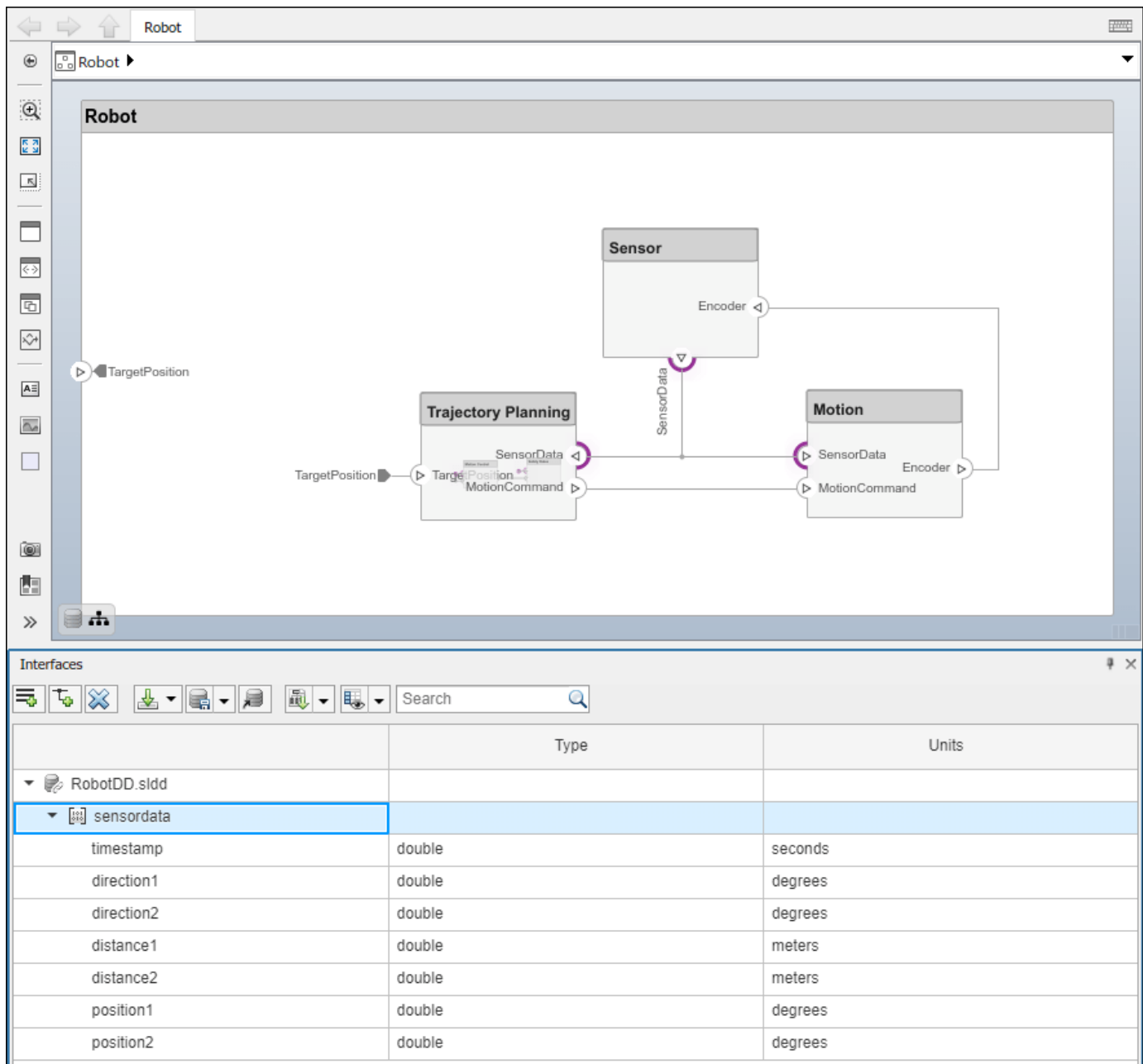
You can simulate the Simulink component implementations in System Composer. Use the **Simulation Data Inspector** to view and compare simulation results between model designs.

Create Simulink Behavior with Robot Arm Model

This example shows how to use a robot arm model to create Simulink® behavior from the `Motion` component.

1. Open the `Robot.slx` model.

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



The Robot model has an interface `sensordata` applied on the ports `SensorData`.

2. Look up the Motion component.

```
motionComp = lookup(model, 'Path', 'Robot/Motion');
```

3. Create a Simulink behavior.

```
motionComp.createSimulinkBehavior('MotionSimulink');
```

Create Referenced Simulink Behavior Model

When a component does not require decomposition from an architecture standpoint, you can design and define the behavior of the component in Simulink. When you link to a Simulink behavior, the Component block becomes a Reference Component block.

A reference component is a component whose definition is a separate architecture model, Simulink behavior model, or Simulink subsystem behavior. A reference component represents a logical hierarchy of other compositions.

You can reuse compositions in the model using reference components. There are three types of reference components:

- *Model references* are Simulink models.
- *Subsystem references* are Simulink subsystems.
- *Architecture references* are System Composer architecture models.

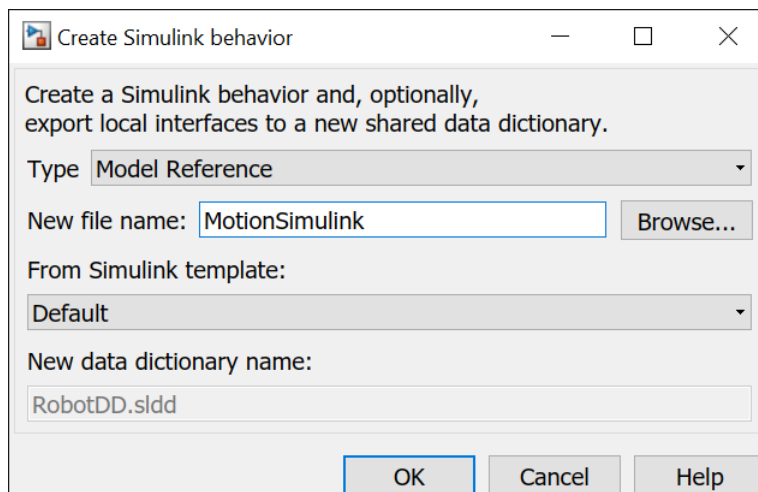
In this section, you will create a model reference and a subsystem reference. For more information on architecture references, see “Create Reference Architecture” on page 1-18.


Referenced or linked models are useful for collaborative design with version control using Simulink projects. For more information, see “Organize System Composer Files in Projects” on page 10-2.

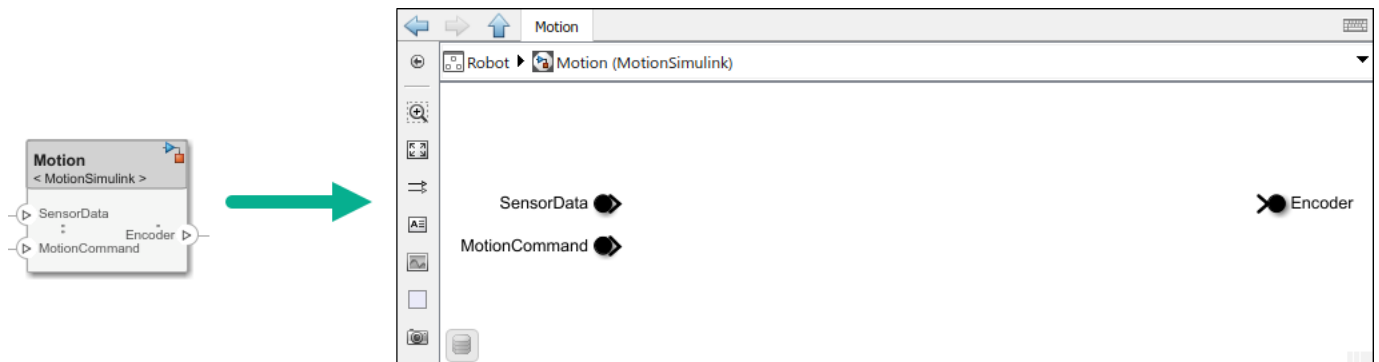
Create Reusable Simulink Behavior Using Model Reference Component

Use Simulink model references to describe the implementation of System Composer components.

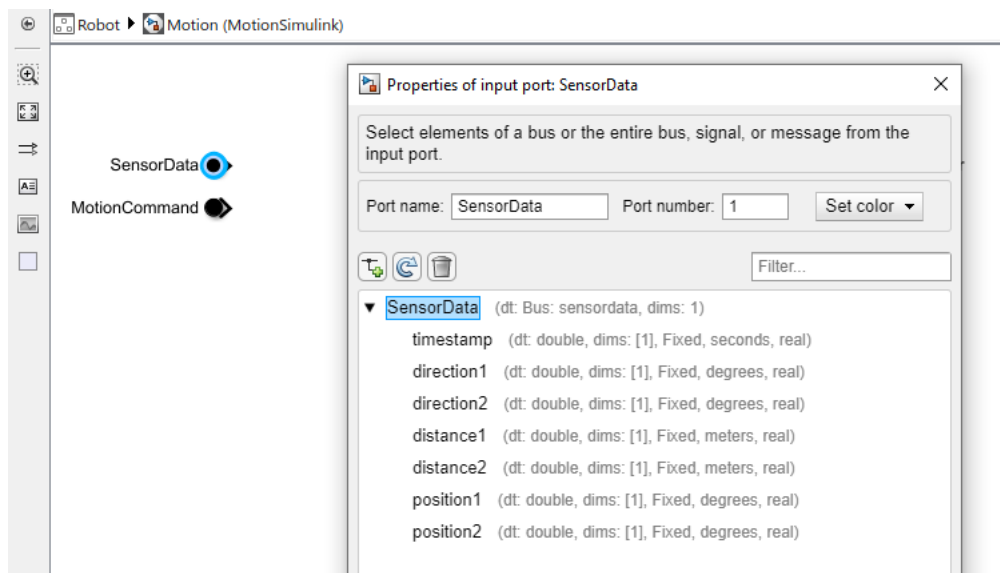
- 1 Right-click the **Motion** component and select **Create Simulink Behavior**. Alternatively, navigate to **Modeling > Create Simulink Behavior**.
- 2 From the **Type** list, select **Model Reference**. Provide the model name **MotionSimulink**. The default name is the name of the component.



- 3 A new Simulink model file with the provided name is created in the current folder. 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  icon on the component indicates that the component has a Simulink behavior.



- 4 To view the interfaces on the SensorData port converted into Simulink bus elements, double-click the port in Simulink.



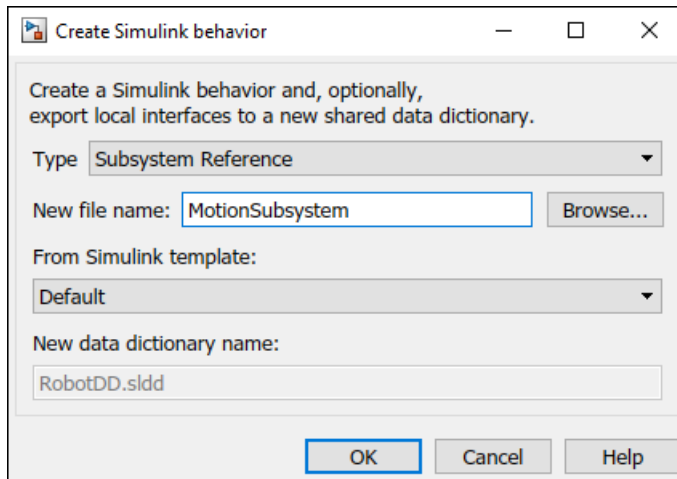
- 5 To remove model behavior, right-click the linked Motion component and select **Inline Model**.


For more information on removing referenced behaviors, see “Remove Reference Architecture” on page 1-20.

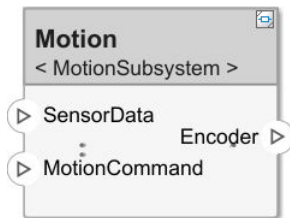
Create Reusable Simulink Subsystem Behavior Using Subsystem Reference Component

Use subsystem references to author Simulink or Simscape behaviors with physical ports, connections, and blocks. For more information, see “Describe Component Behavior Using Simscape” on page 5-75.

- 1 Right-click the Motion component and select **Create Simulink Behavior**. Alternatively, navigate to **Modeling > Create Simulink Behavior**.
- 2 From the **Type** list, select **Subsystem Reference**. Provide the model name MotionSubsystem. The default name is the name of the component.



- 3 A new Simulink subsystem file with the provided name is created in the current folder. The root-level ports of the Simulink subsystem reflect the ports of the component. The component in the architecture model is linked to the Simulink subsystem. The  icon on the component indicates that the component has a Simulink subsystem behavior.



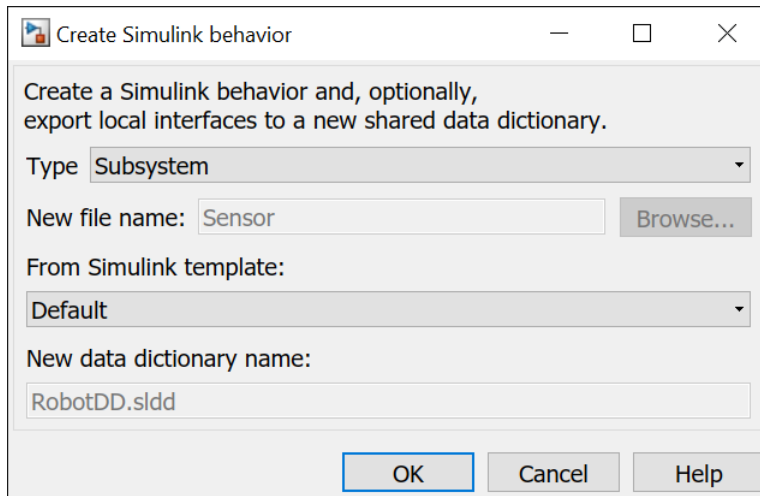
You can access and edit referenced Simulink models and subsystems by double-clicking the Reference Component in the architecture model. When you save the architecture model, all unsaved referenced Simulink behaviors are also saved, and all linked components are updated.

Create Simulink Subsystem Behavior Using Subsystem Component


A subsystem component is a Simulink subsystem that is part of the parent System Composer architecture model.

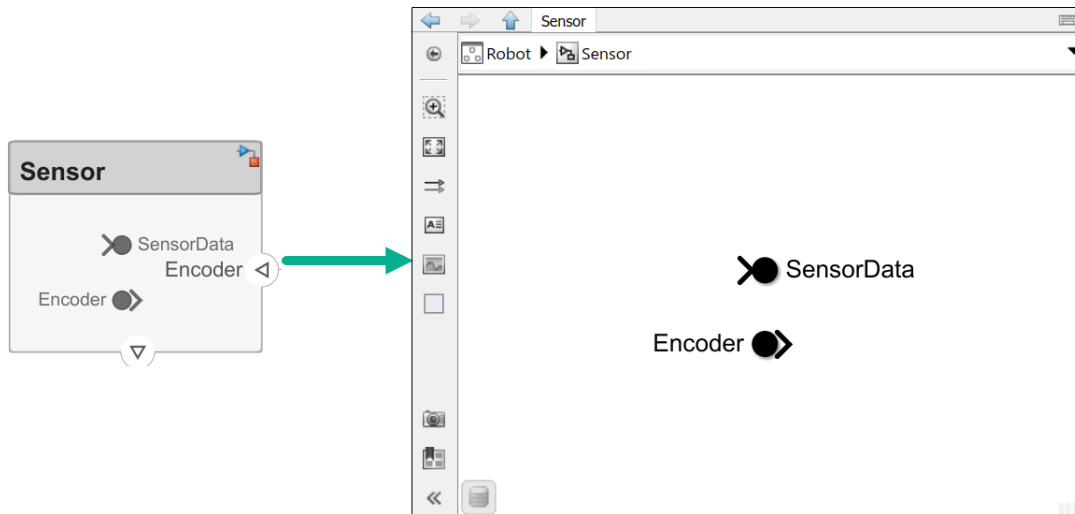
Add Simulink subsystem behavior to a component to author a subsystem component in System Composer. You cannot synchronize and reuse subsystem components as Reference Component blocks because the component is part of the parent model.

- 1 Right-click the Sensor component and select **Create Simulink Behavior**. Alternatively, navigate to **Modeling > Create Simulink Behavior**.
- 2 From the **Type** list, select **Subsystem**.

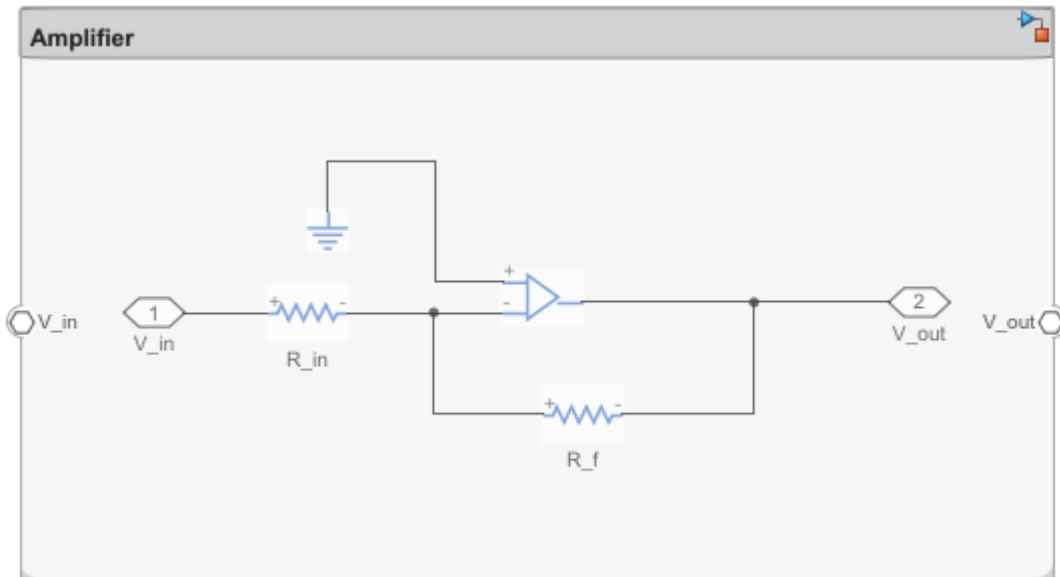


- 3 The Sensor component is now a Simulink subsystem of the same name that is part of the parent System Composer architecture model.

The root-level ports of the Simulink model reflect the ports of the component. The  icon on the component indicates that the component has a Simulink subsystem behavior.



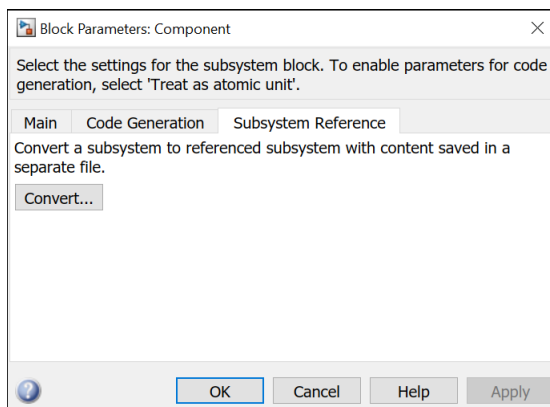
- 4 You can continue to provide specific dynamics and algorithms in the Simulink subsystem behavior model. Adding root-level ports in the subsystem behavior creates additional ports on the subsystem component.
- 5 You can use subsystem components to author Simscape component behaviors with physical ports, connections, and blocks. For example, this amplifier physical system uses electrical domain blocks inside a subsystem component in a System Composer architecture model.



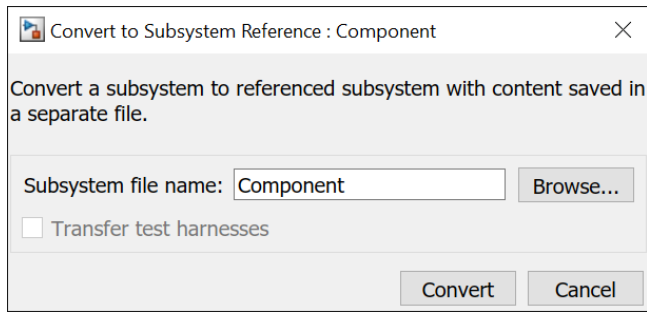
Convert Simulink Subsystem Component to Subsystem Reference Component

You can convert existing Simulink subsystem components that are part of the parent System Composer model to subsystem reference components. The subsystem reference components are saved separately as a reusable artifact.

- 1 Right-click the subsystem component block and select **Block Parameters (Subsystem)**.
- 2 Click the **Subsystem Reference** tab.



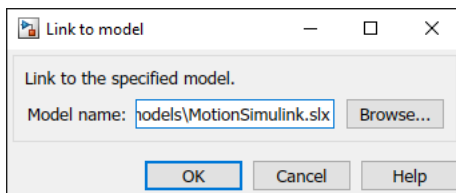
- 3 Click **Convert** to open the **Convert to Subsystem Reference** dialog.
- 4 Choose a name for the new subsystem file. Optionally, select **Transfer test harnesses** to transfer test harnesses. Click **Convert** to complete the conversion.



To convert a subsystem component to a subsystem reference programmatically, use the `createSimulinkBehavior` function.

Link to Existing Simulink Behavior Model

You can link to an existing Simulink behavior model or subsystem 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 or subsystem.



Any subcomponents and ports in the components are deleted when the component links to a Simulink model or subsystem. A prompt displays to continue and lose subcomponents and ports.

Note Linking a System Composer component to a Simulink model with root-level enable or trigger ports is not supported.

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. The change is reflected when you refresh the model.

Access Model Arguments as Parameters on Reference Components

System Composer exposes instance-specific parameter values for reusable referenced models.

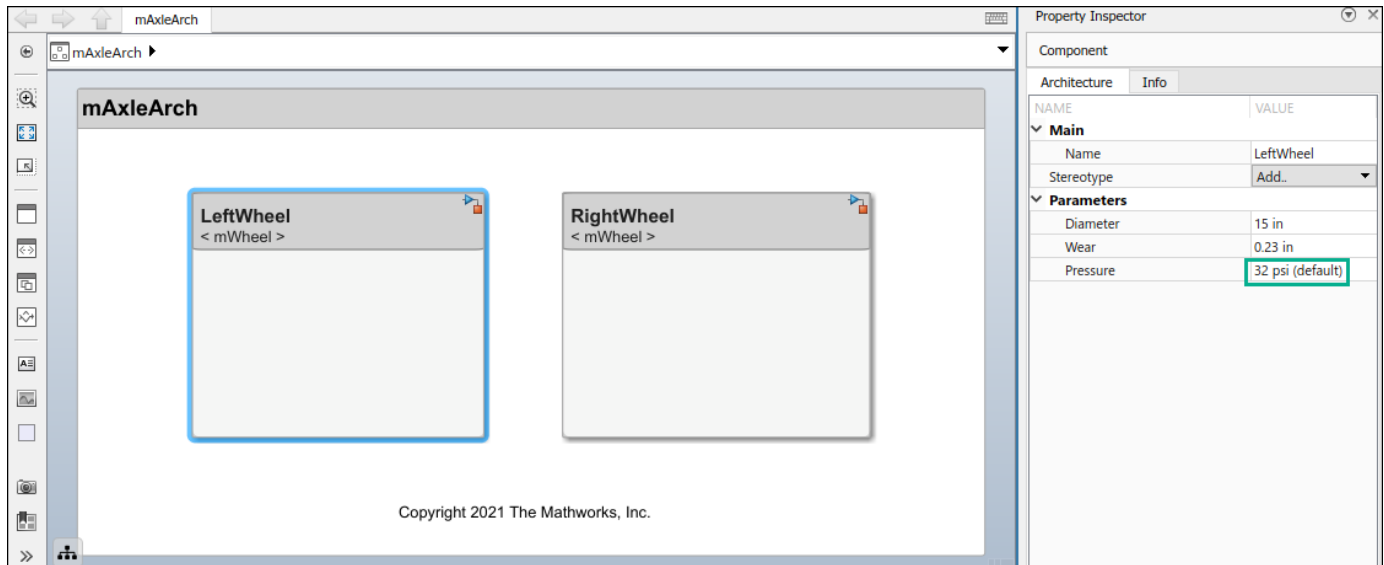
A parameter definition is the definition of a property that has instance semantics. A parameter definition specifies attributes such as name, data type, default value, and units.

Parameter definitions can be specified as model arguments on a Simulink model or a System Composer architecture model.

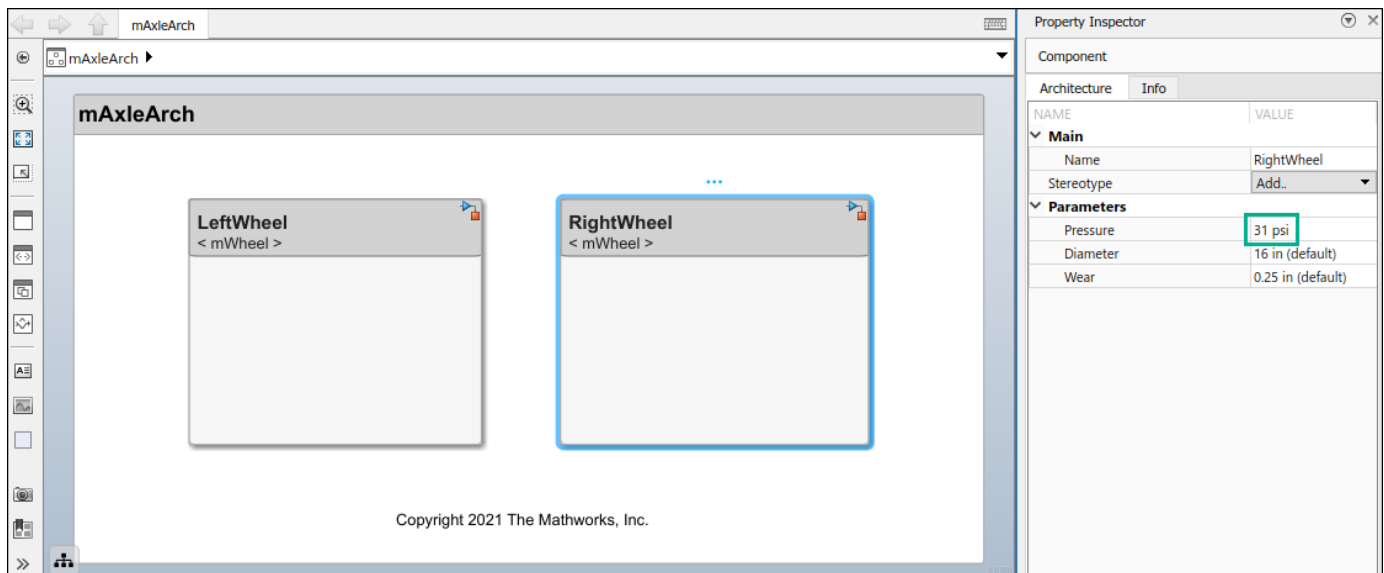
A parameter is an instance-specific value of a parameter definition. A parameter captures instance-specific values and units.

Parameters are available for each component linking to a model reference or architecture reference that specifies model arguments. You can specify independent values for a parameter on each component.

Instance-specific parameter values are visible on the component level. View and edit these values using the Property Inspector.



Each parameter value can be specified independently for each component that references the model.

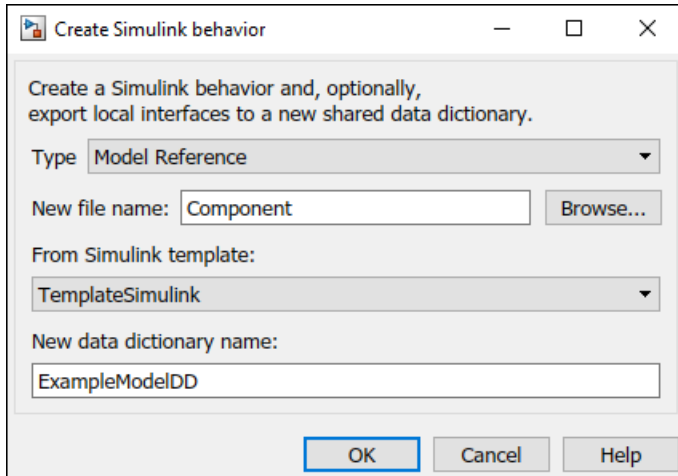


For more information, see “Use Parameters to Store Instance Values with Components” on page 5-13.

Create Simulink Behavior from Template for Component

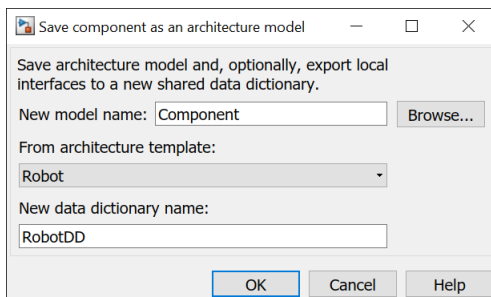
To create user-defined templates for Simulink models, see “Create Template from Model”.

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, navigate to **Modeling > Create Simulink Behavior**.



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 that you can use architecture templates by right-clicking a component and selecting **Save As Architecture Model**, or navigating to **Modeling > Save As Architecture Model**.



See Also

Functions

`createSimulinkBehavior` | `linkToModel` | `createArchitectureModel` | `systemcomposer.parameter.ParameterDefinition`

Blocks

Reference Component

More About

- “Decompose and Reuse Components” on page 1-17
- “Describe Component Behavior Using Stateflow Charts” on page 5-23

- “Describe Component Behavior Using Simscape” on page 5-75
- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Organize System Composer Files in Projects” on page 10-2

Use Parameters to Store Instance Values with Components

This example shows how to add parameter definitions as model arguments to a System Composer™ architecture model of a wheel, `mWheelArch.slx`, using the **Model Explorer**. Then, on the System Composer architecture model `mAxleArch.slx`, these model arguments are exposed as instance-specific parameter values that can be changed independently across each component that references `mWheelArch`.

Use Model Explorer to Add MATLAB Variables as Model Arguments

Open the `mWheelArch` model.

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

Navigate to **Modeling > Model Explorer** or enter **Ctrl+H**. The **Model Explorer** opens. Expand the `mWheelArch` model and select **Model Workspace**. View the contents of the model workspace. The workspace contains three Simulink® parameters named `Diameter`, `Pressure`, and `Wear`.

Column View:		Data Objects	Show Details		3 object(s)					
	Name	Value	DataType	Dimensions	Complexity	Min	Max	Unit	Argument	StorageClass
	Diameter	16	double	[1 1]	real	[]	[]	in	<input checked="" type="checkbox"/>	Configure
	Pressure	32	double	[1 1]	real	[]	[]	psi	<input checked="" type="checkbox"/>	Configure
	Wear	0.25	double	[1 1]	real	[]	[]	in	<input checked="" type="checkbox"/>	Configure

To add a new MATLAB® variable to the model workspace, on the toolbar menu, click . You can rename your variable from the default name `Var` and set its starting value. If you select the **Argument** check box, the MATLAB variable becomes a model argument. As a model argument, the variable can later be exposed as an instance-specific parameter value in an architecture model. Rename the variable to `TreadDepth` and set its value to `1`, then select it as a model argument.

Column View:		Data Objects	Show Details		4 object(s)					
	Name	Value	DataType	Dimensions	Complexity	Min	Max	Unit	Argument	StorageClass
	Diameter	16	double	[1 1]	real	[]	[]	in	<input checked="" type="checkbox"/>	Configure
	Pressure	32	double	[1 1]	real	[]	[]	psi	<input checked="" type="checkbox"/>	Configure
	Wear	0.25	double	[1 1]	real	[]	[]	in	<input checked="" type="checkbox"/>	Configure
	TreadDepth	1	double (auto)	[1 1]	real				<input checked="" type="checkbox"/>	

Use Model Explorer to Add Simulink Parameters as Model Arguments

In the Model Explorer, you can also add Simulink parameters as model arguments. To add a new

Simulink parameter to the model workspace, on the toolbar menu, click . You can edit the attributes of a Simulink parameter including: Name, Value, DataType, Dimensions, Complexity, Min, Max, and Unit. These attributes contribute to the parameter definition when the parameter is specified as a model argument. Select the **Argument** check box to specify a parameter as a model

argument. Rename the variable to `PressureBar`, set its value to 2000, set its units to mbar, then select it as a model argument.

	Name	Value	Data Type	Dimensions	Complexity	Min	Max	Unit	Argument	Storage Class
[101 010]	Diameter	16	double	[1 1]	real	[]	[]	in	<input checked="" type="checkbox"/>	Configure
[101 010]	Pressure	32	double	[1 1]	real	[]	[]	psi	<input checked="" type="checkbox"/>	Configure
[101 010]	Wear	0.25	double	[1 1]	real	[]	[]	in	<input checked="" type="checkbox"/>	Configure
[101 010]	TreadDepth	1	double (auto)	[1 1]	real				<input checked="" type="checkbox"/>	
[101 010]	PressureBar	2000	auto	[1 1]	real	[]	[]	mbar	<input checked="" type="checkbox"/>	Configure

Right-click the `mWheel` model in the **Model Explorer**. Save these changes to the model workspace, then close the **Model Explorer**.

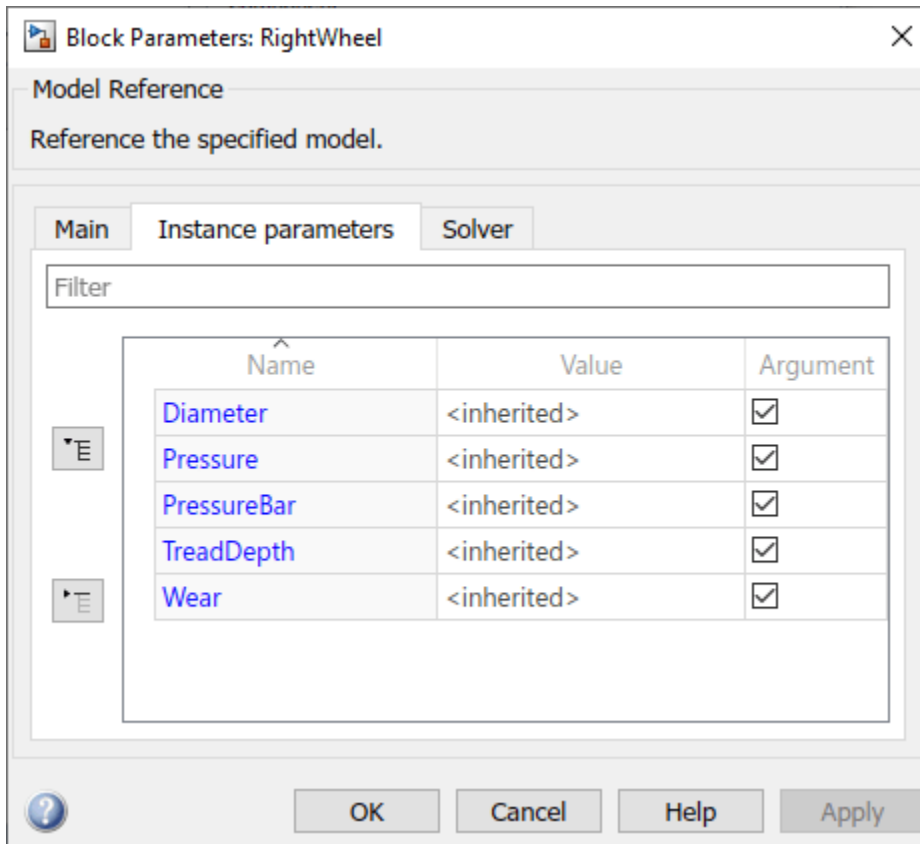
View and Edit Parameters on Components in Architecture Model

Open the `mAxleArch` architecture model.

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

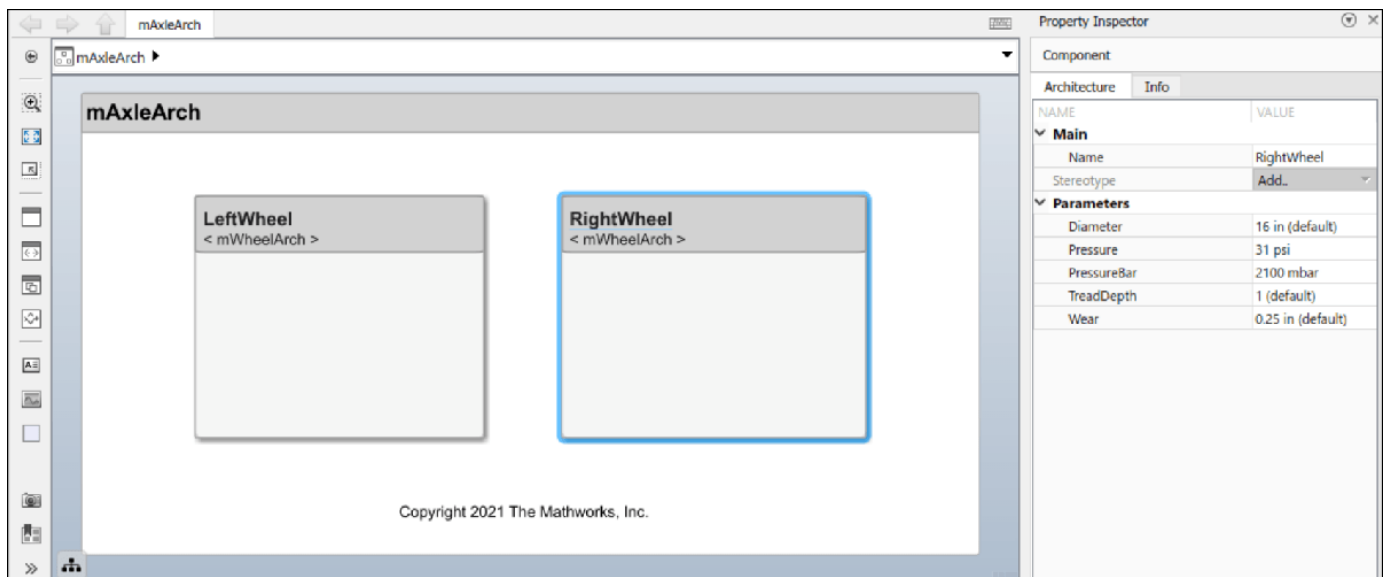
Select the `LeftWheel` component that references the `mWheel` model. The parameters appear on the Property Inspector with default values.

You can expose these parameters as model arguments and then edit the parameters as instance-specific parameters independently for each component that references the same model. Right-click the `RightWheel` component and select **Block Parameters (Model Reference)**. Click the **Instance parameters** tab and select the **Argument** check box for the new parameters `Pressure` and `TreadDepth`.



Once selected, these parameters are treated as model arguments of the `mAxleArch` model and can be changed independently for each instance the model.

Edit the parameters for the `RightWheel` component so that `Pressure` and `PressureBar` are now 31 psi and 2100 mbar, respectively.



The corresponding parameter values for the `LeftWheel` component remain unchanged.

See Also

`systemcomposer.parameter.ParameterDefinition` | `getParameterDefinition` | `getEvaluatedParameterValue` | `getParameterNames` | `getParameterValue` | `setParameterValue` | `setUnit`

More About

- “Describe Component Behavior Using Simulink” on page 5-2
- “Compose Architecture Visually” on page 1-2
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22
- “Modeling System Architecture of Small UAV” on page 1-32
- “Organize System Composer Files in Projects” on page 10-2

Extract Architecture of Simulink Model Using System Composer

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

Convert Simulink Model to System Composer Architecture

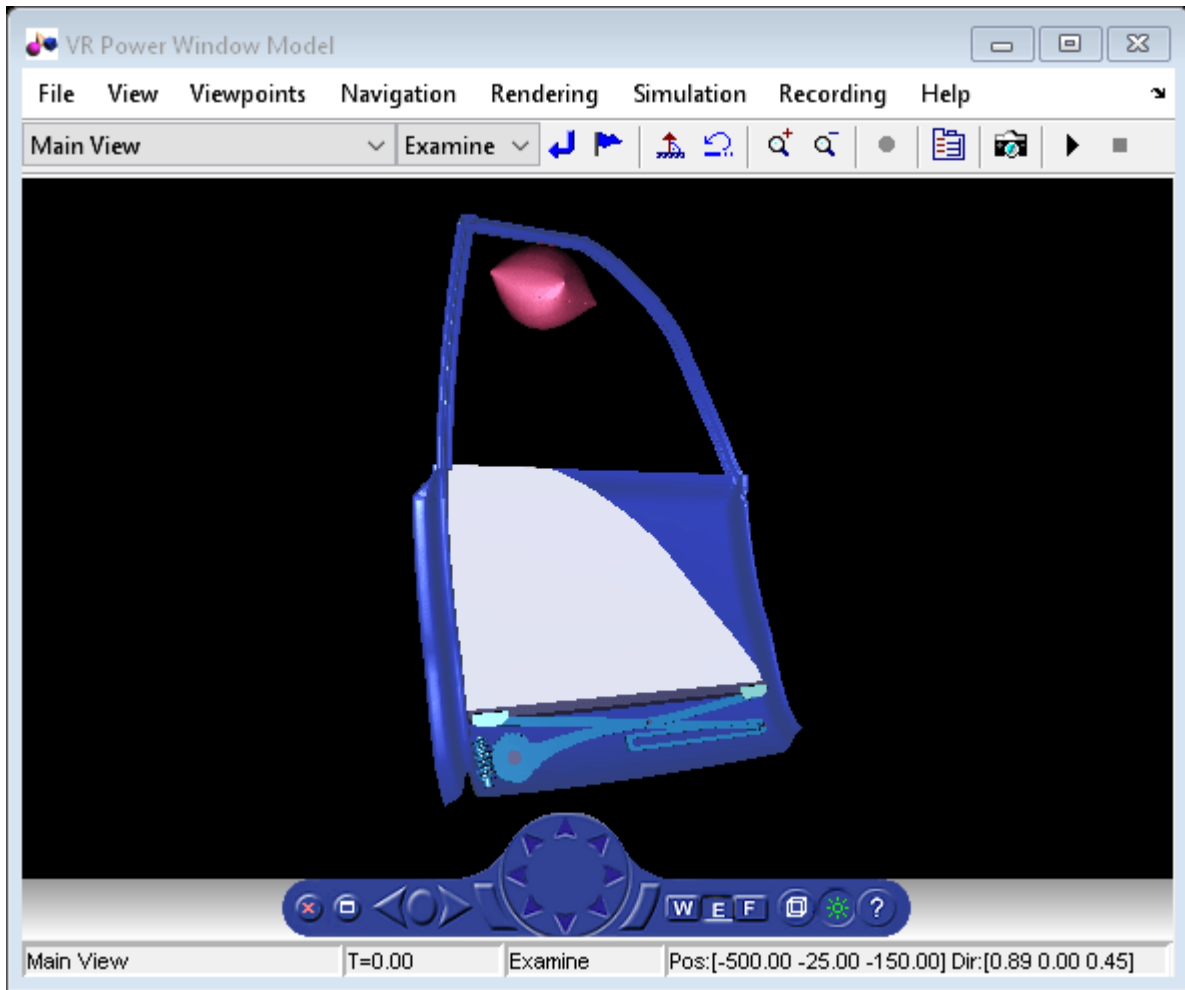
System Composer converts structural constructs in a Simulink model to equivalent architecture model constructs:

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

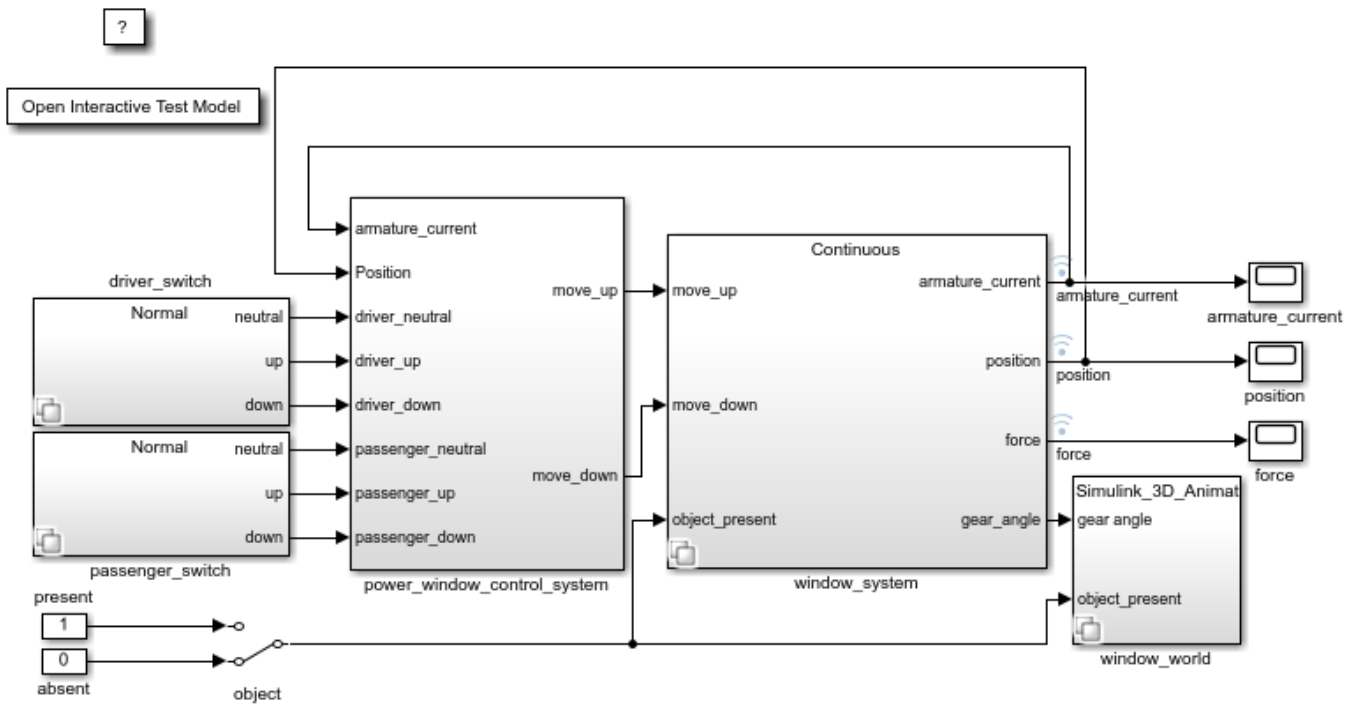
Open the Model

Open the Simulink model of the VR Power Window Model.

```
slexPowerWindowStart
```



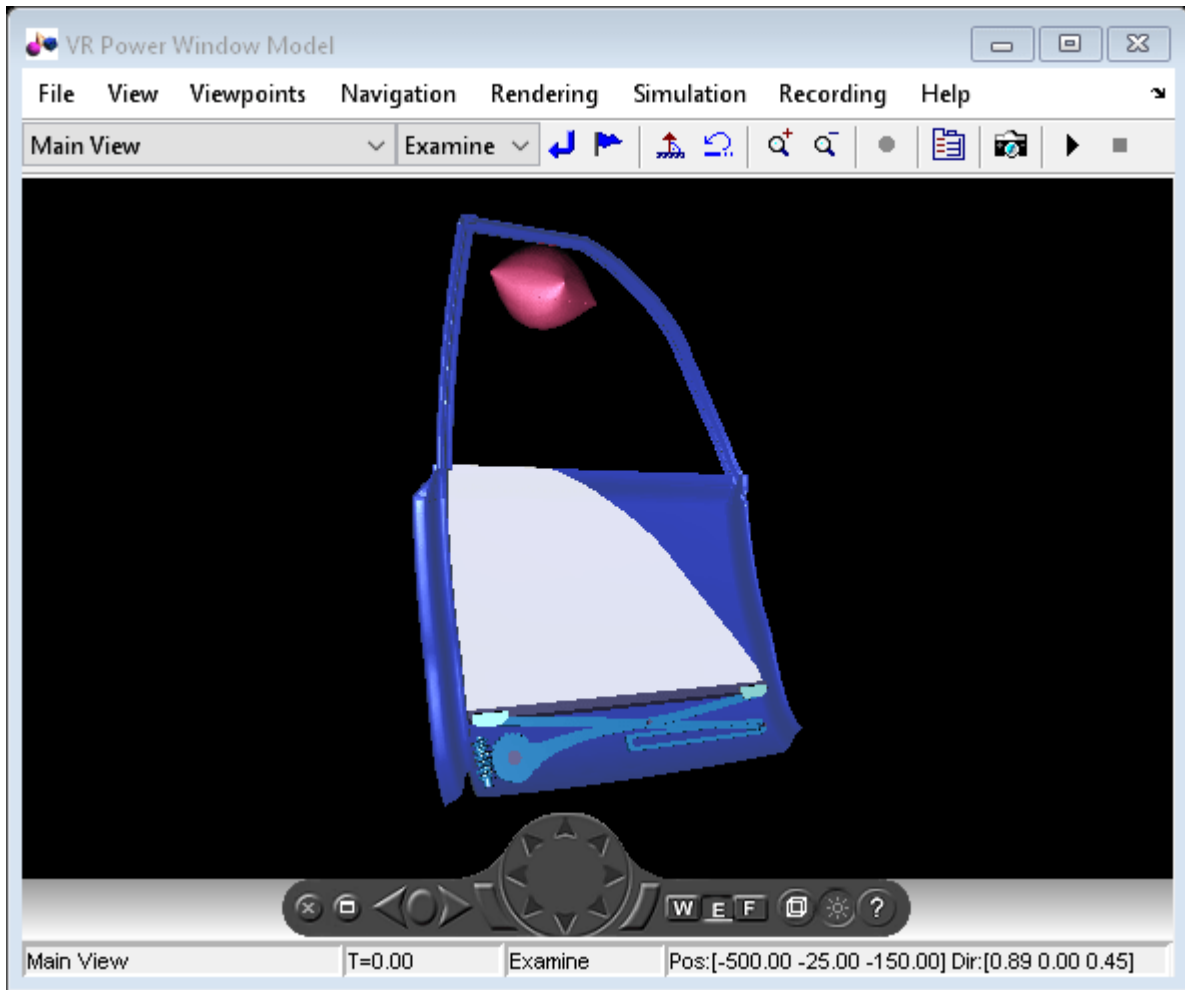
```
open_system('slexPowerWindowExample');
```



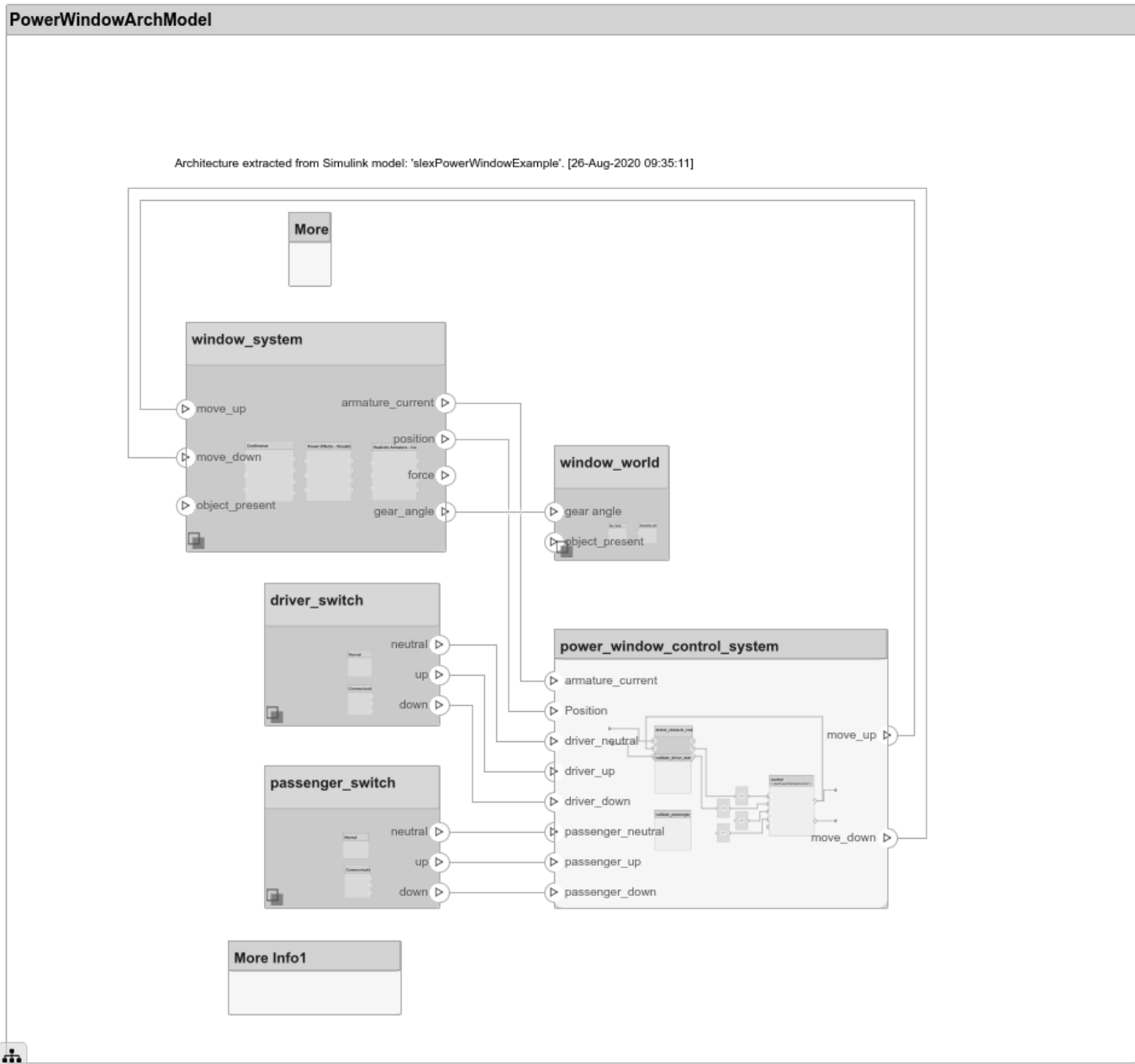
Export the Model

Extract an architecture model from the original model.

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



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



See Also

[extractArchitectureFromSimulink](#)

More About

- “Extract Architecture from Simulink Model” on page 5-28
- “Compose Architecture Visually” on page 1-2
- “Describe Component Behavior Using Simulink” on page 5-2

- “Simulate Mobile Robot with System Composer Workflow” on page 4-22
- “Modeling System Architecture of Small UAV” on page 1-32

Describe Component Behavior Using Stateflow Charts

A state chart diagram demonstrates the state-dependent behavior of a component throughout its state lifecycle and the events that can trigger a transition between states.

Add Stateflow chart behavior to describe a component using state machines. You cannot synchronize and reuse Stateflow chart behaviors as Reference Component blocks because the component is part of the parent model.

You can simulate the Stateflow component implementations in System Composer. Use the **Simulation Data Inspector** to view and compare simulation results between model designs.

State charts consist of a finite set of states with transitions between them to capture the modes of operation for the component. Charts allow you to 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. This functionality requires a Stateflow license. For more information, see “Stateflow”.

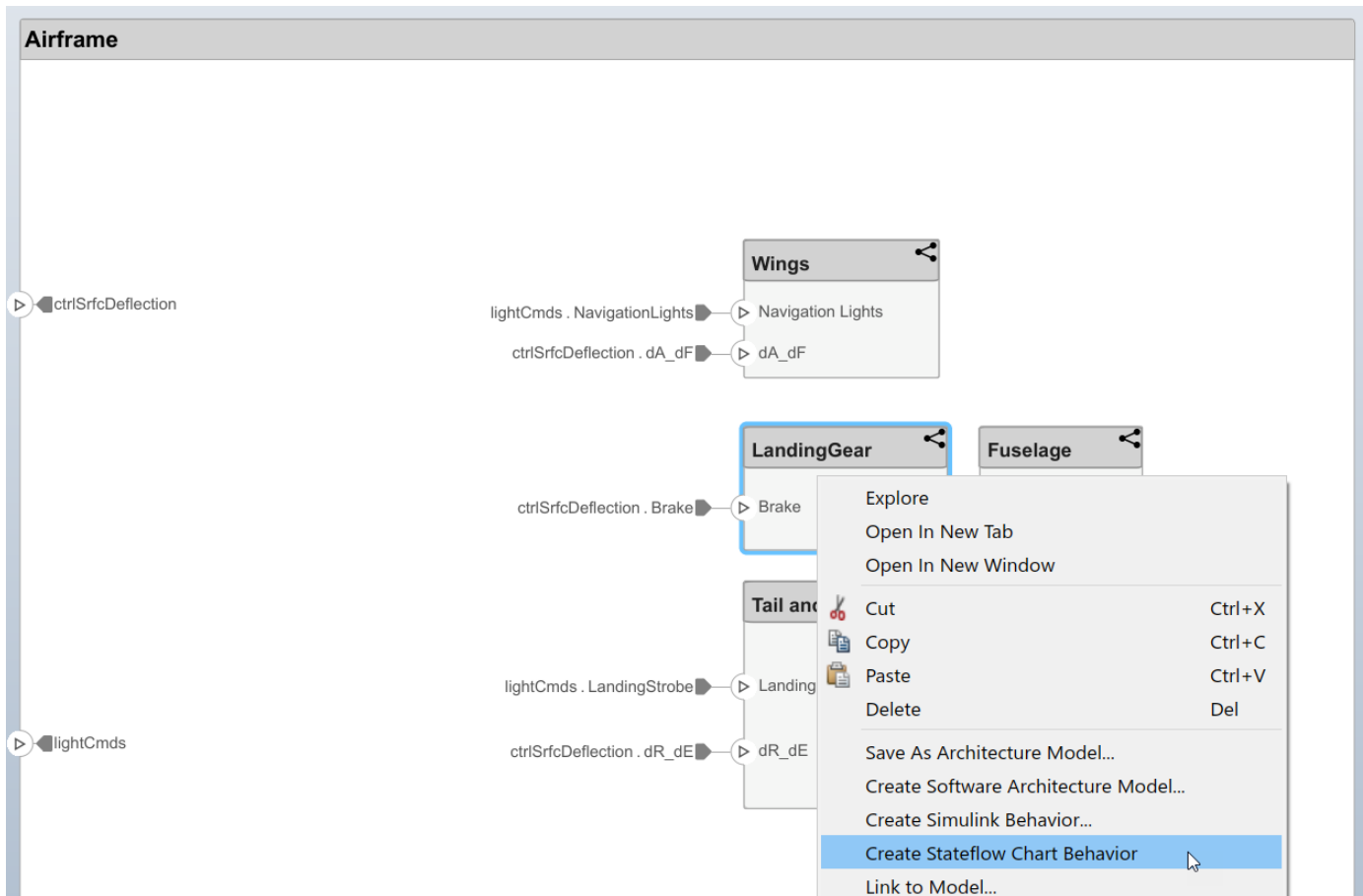
Add State Chart Behavior to Component

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. In the MATLAB Command Window, enter the following command:

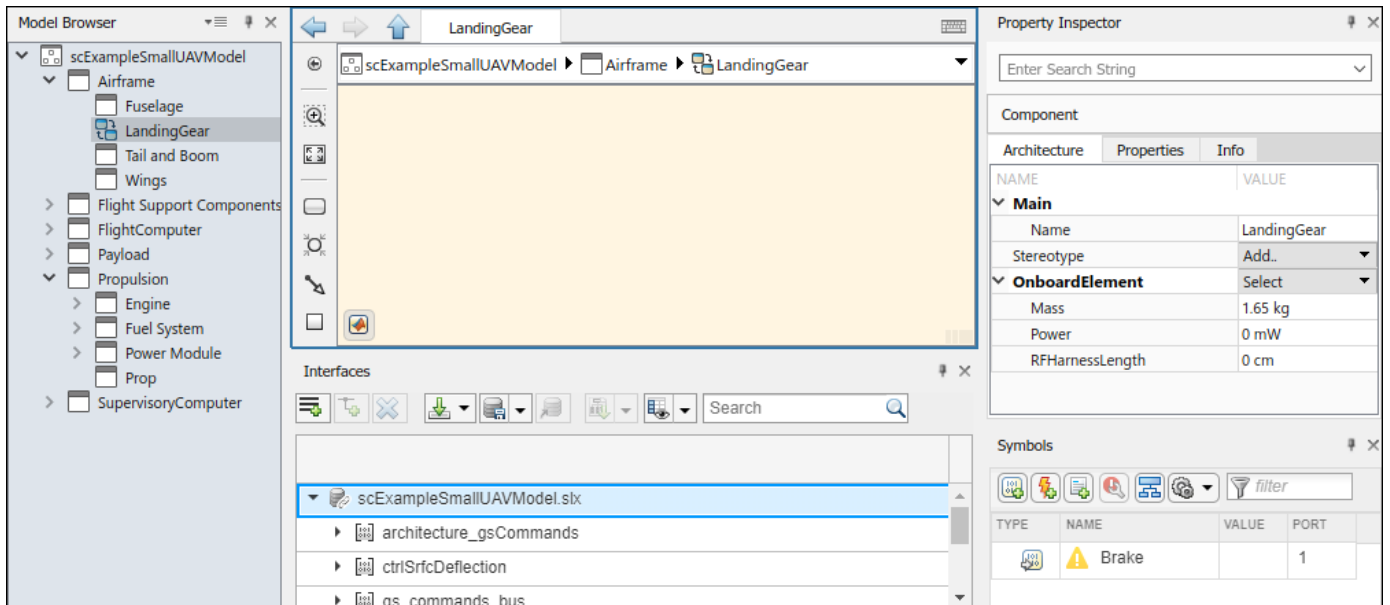
```
scExampleSmallUAV
```

- 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 **Modeling > Interface Editor**. Right-click the interface operator `Cmds` and select **Assign to Selected Port(s)**.
- 4 Right-click the `LandingGear` component and select **Create Stateflow Chart Behavior**. Alternatively, navigate to **Modeling > Create Stateflow Chart Behavior**.



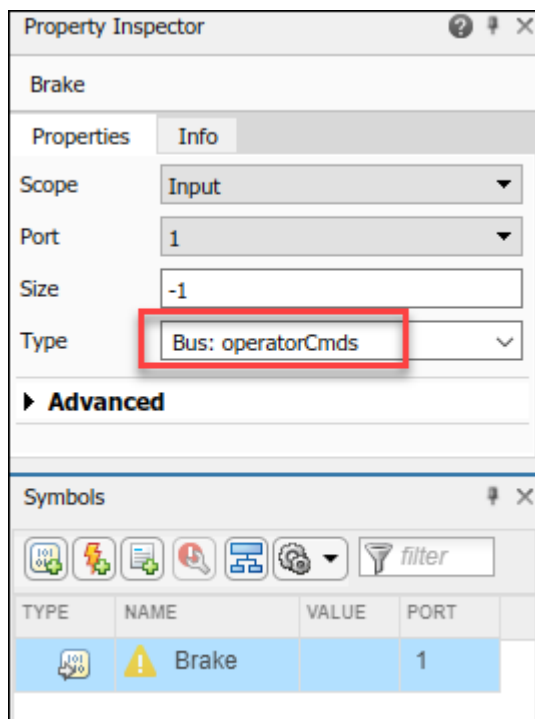
- 5 Double-click **LandingGear**, which has the Stateflow icon. Navigate to **Modeling > Design Data > Symbols Pane** to view the Stateflow symbols. The input port **Brake** appears as input data in the symbols pane.

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.

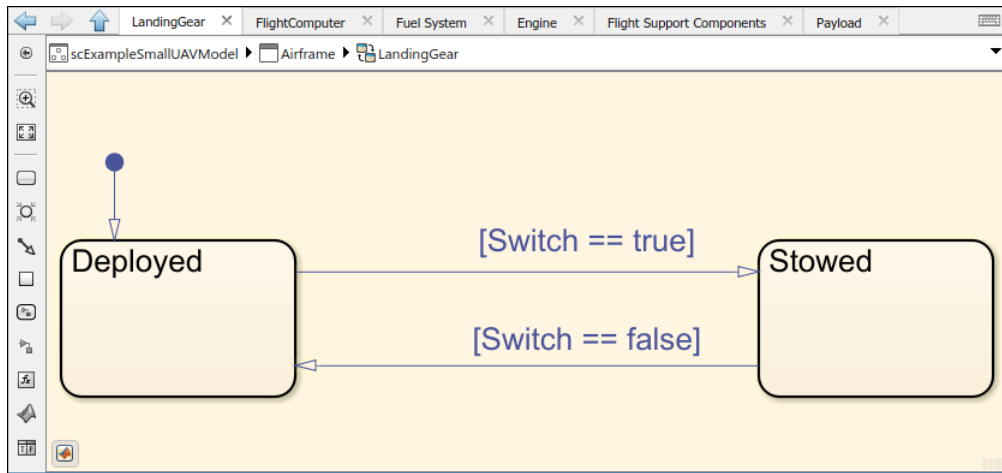


Since Stateflow ports show up as input and output data objects, they must follow Stateflow naming conventions. Ports are automatically renamed to follow Stateflow naming conventions. For more information, see “Guidelines for Naming Stateflow Objects” (Stateflow).

- 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).



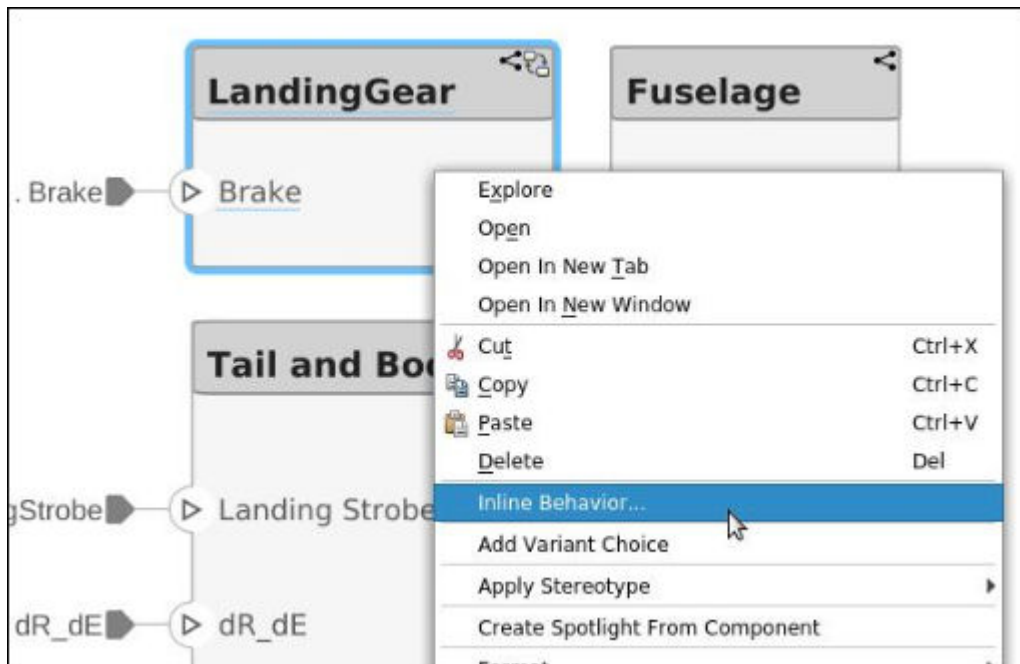
- 7 You can populate the Stateflow canvas to represent the internal states of the LandingGear.



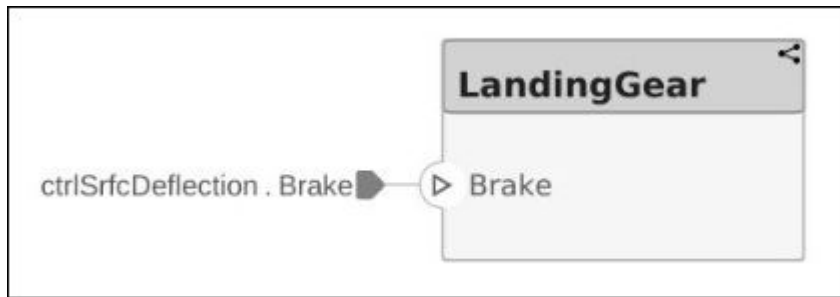
Remove Stateflow Chart Behavior from Component

You can remove Stateflow chart behavior from a component to delete the contents inside the Stateflow chart while preserving interfaces on the component.

- 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, click **OK**.
- 3 The Stateflow chart behavior on the component is removed. Interfaces on the component are preserved.



See Also

`createStateflowChartBehavior | inlineComponent`

More About

- “Compose Architecture Visually” on page 1-2
- “Decompose and Reuse Components” on page 1-17
- “Describe Component Behavior Using Simulink” on page 5-2
- “Describe Component Behavior Using Simscape” on page 5-75
- “Extract Architecture from Simulink Model” on page 5-28
- “Describe System Behavior Using Sequence Diagrams” on page 5-32

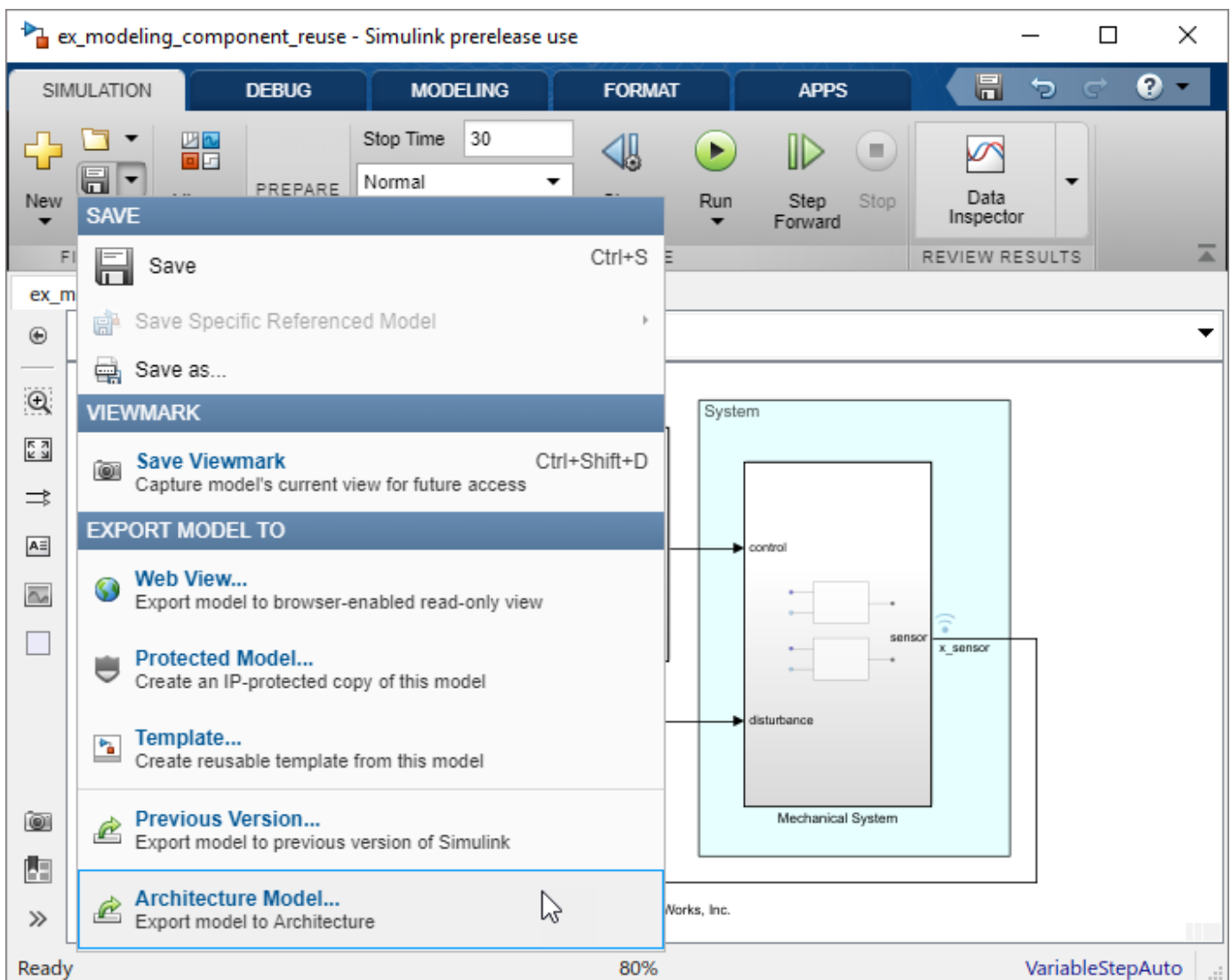
Extract Architecture from Simulink Model

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.

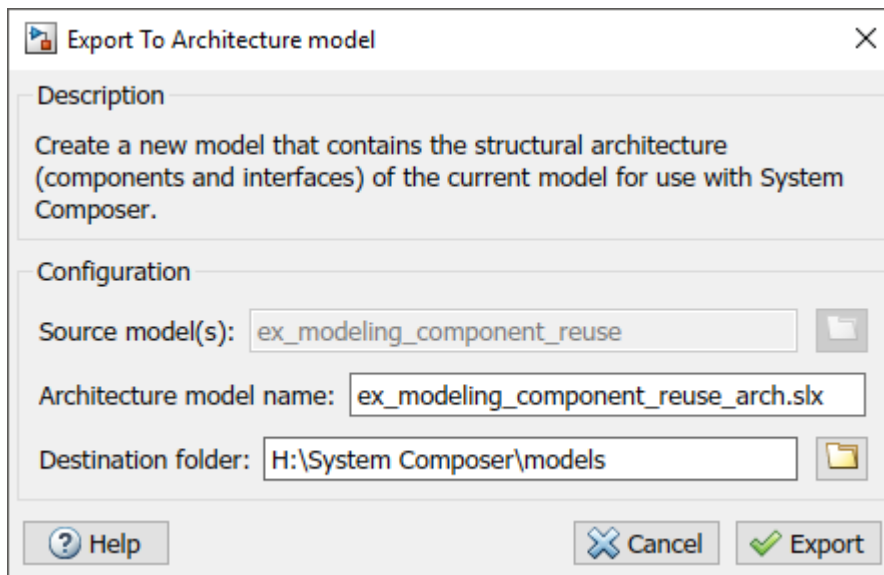
- 1 Open an example model.

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

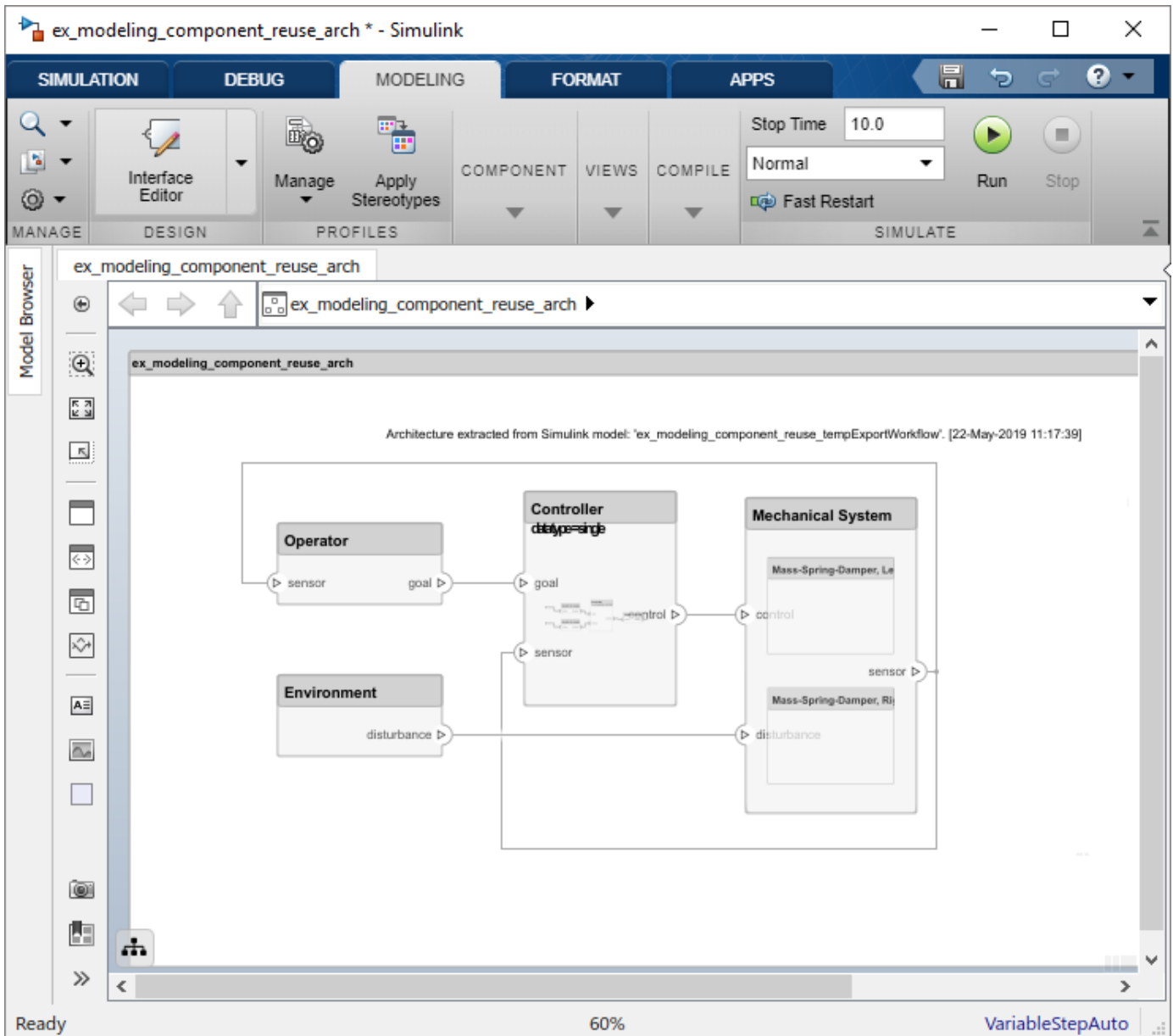
- 2 On the **Simulation** tab, click the **Save** arrow. From the **Export Model To** list, select **Architecture Model**.



- 3 Provide a name and path for the architecture model.



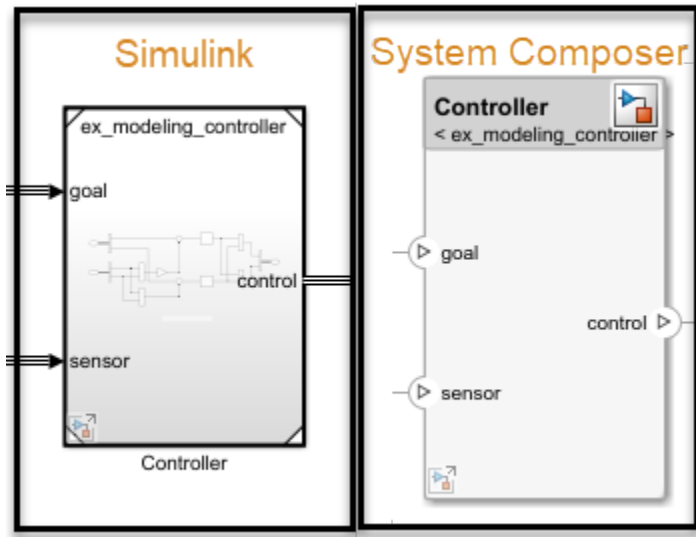
- 4 Click **Export**. A System Composer Editor window opens with an architecture model corresponding to the Simulink model.



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 Architecture of Simulink Model Using System Composer” on page 5-17
- “Describe Component Behavior Using Simulink” on page 5-2
- “Describe Component Behavior Using Stateflow Charts” on page 5-23
- “Describe Component Behavior Using Simscape” on page 5-75
- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Compose Architecture Visually” on page 1-2

Describe System Behavior Using 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.

Use sequence diagrams to describe how the parts of a static system interact.

Sequence diagrams are integrated with architecture models in System Composer.

- “Author Sequence Diagram for Traffic Light Example” on page 5-38: Interactively create and edit a sequence diagram and learn terminology.
- “Implement Sequence Diagram Fragments” on page 5-65: Learn how to implement fragments and more about fragment semantics.
- “Use Sequence Diagrams with Architecture Models” on page 5-51: Create and use sequence diagrams with architecture models.
- “Synchronize Sequence Diagrams and Architecture Models” on page 5-81: Learn how to synchronize sequence diagrams with architecture models.

Note This example uses Stateflow blocks. If you do not have a Stateflow license, you can open and simulate the model but can only make basic changes, such as modifying block parameters.

Traffic Light Example for Sequence Diagrams

This traffic light example contains sequence diagrams to describe pedestrians crossing an intersection. The model describes these steps:

- 1 The traffic signal cycles from red to yellow to green.
- 2 When the pedestrian crossing button is pressed, if the traffic signal is green, the traffic signal transitions from yellow to red for a limited time.
- 3 The pedestrians cross while the walk signal is active.

Open the System Composer model that contains the sequence diagrams.

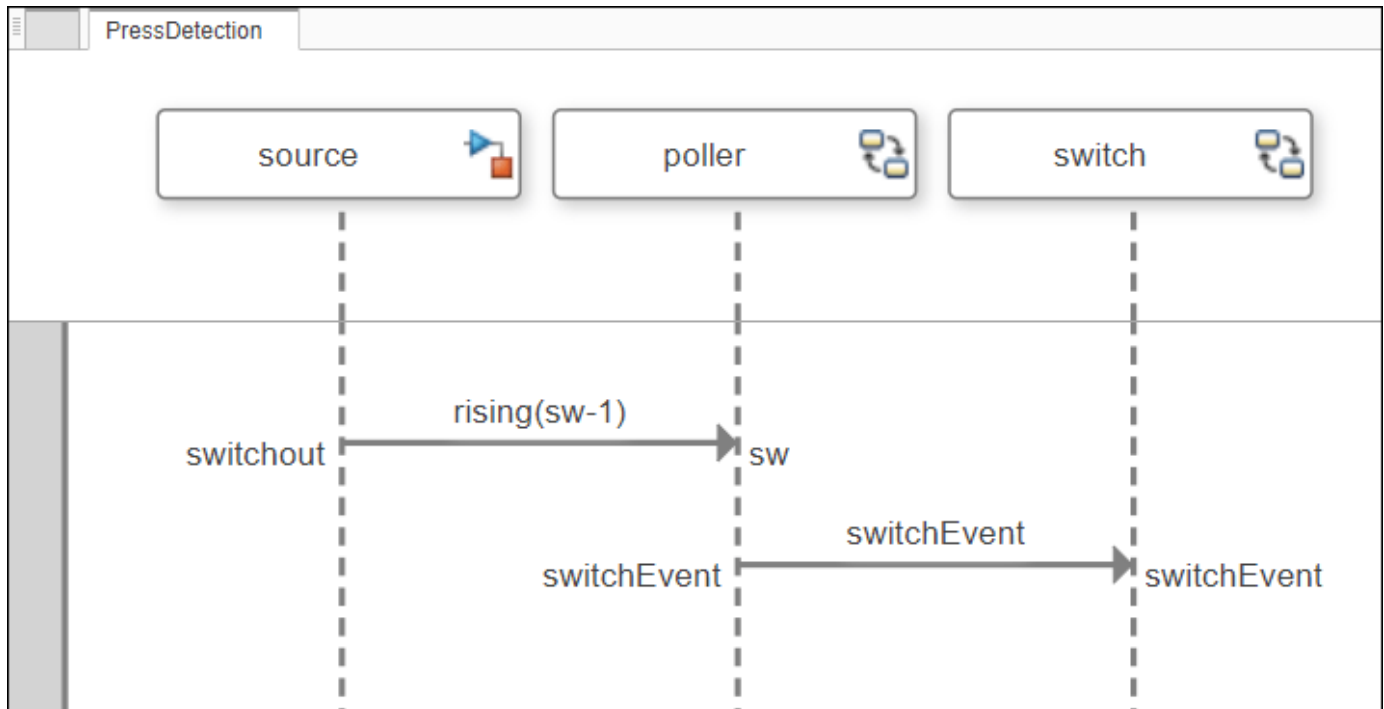
```
model = systemcomposer.openModel('TLExample');
```

Open the Architecture Views Gallery to view the sequence diagrams.

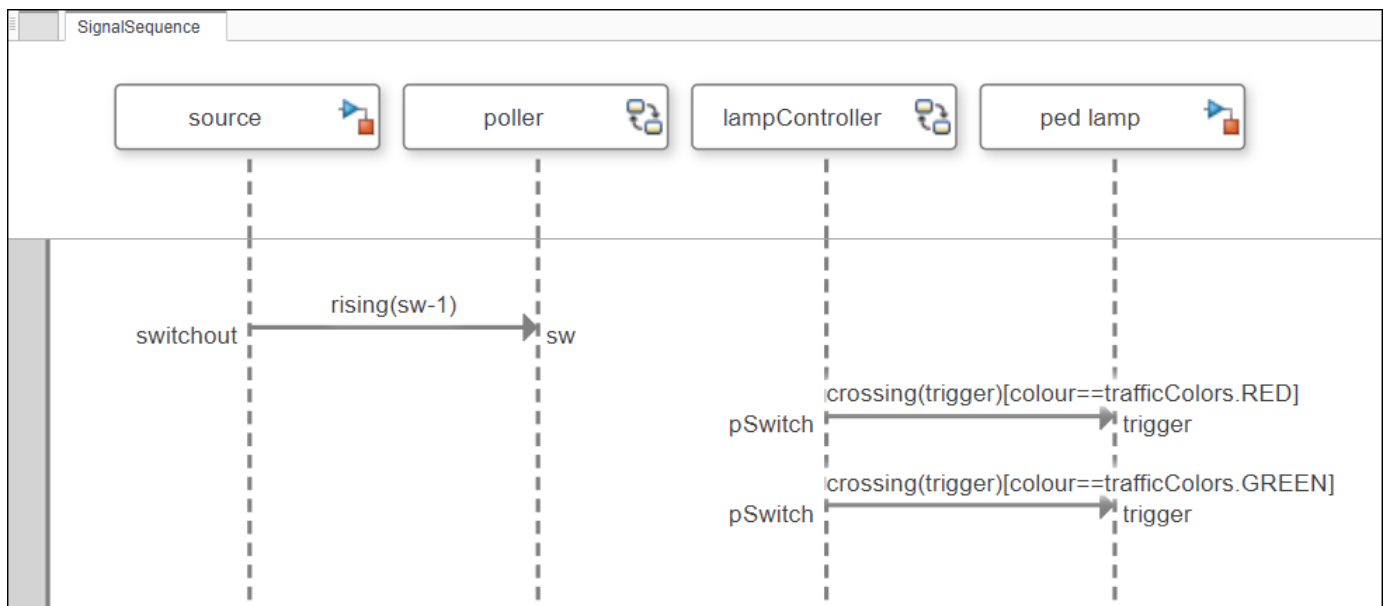
```
openViews(model)
```

The sequence diagrams in this example represent operative scenarios in the architecture model.

1. **PressDetection** sequence diagram: The pedestrian presses the pedestrian crossing button and the signal `sw` rises to 1. The `poller` lifeline is activated, and a `switchEvent` message occurs on the `switch` lifeline to change the traffic signals to allow the pedestrian to cross.

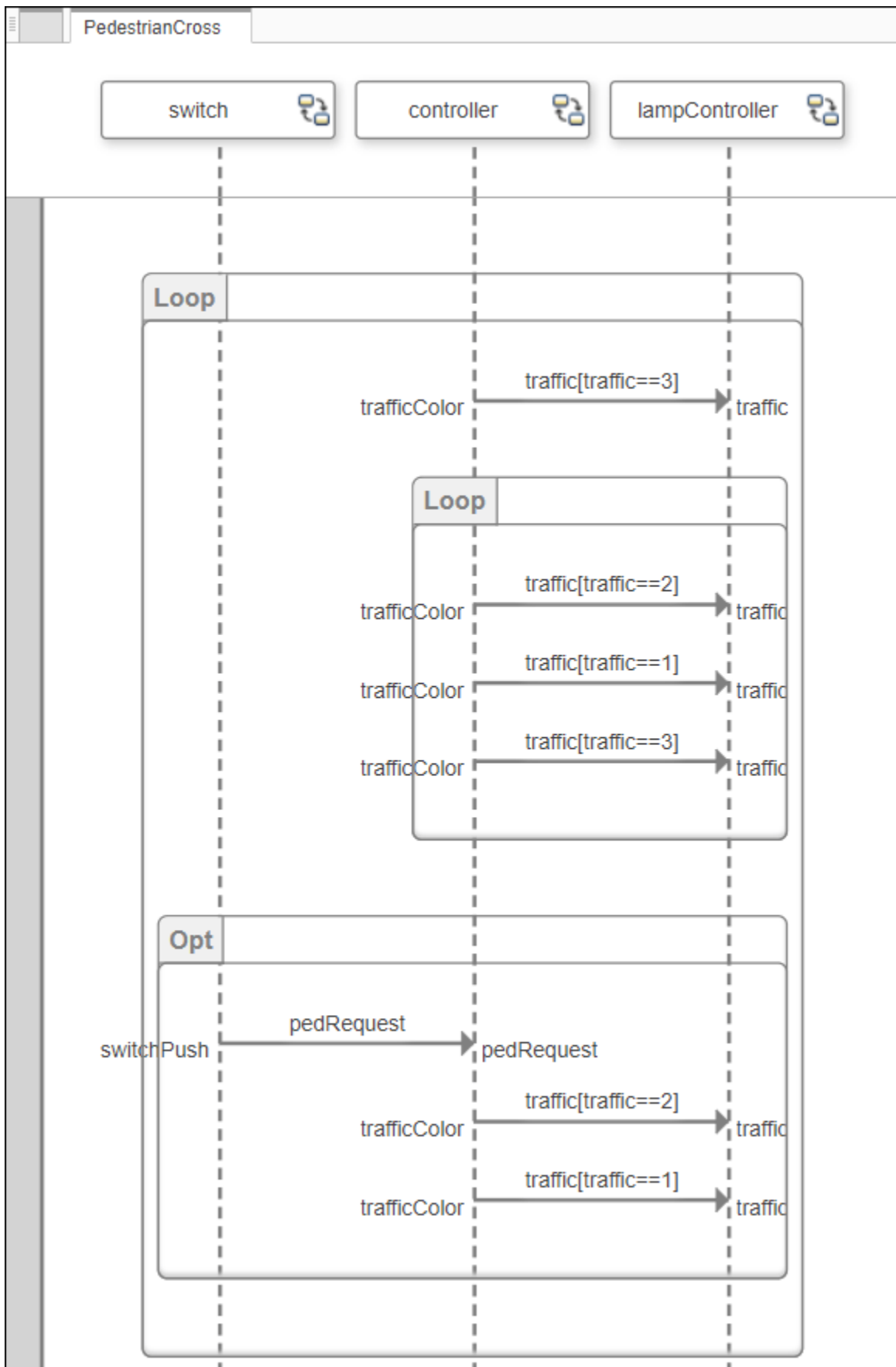


2. SignalSequence sequence diagram: The pedestrian presses the pedestrian crossing button, and the signal `sw` rises to 1. After some intermediary events, the `lampController` lifeline transmits a trigger signal to the `ped lamp` lifeline to change pedestrian lamp traffic colors from RED (stop) to GREEN (go), allowing pedestrians to cross.

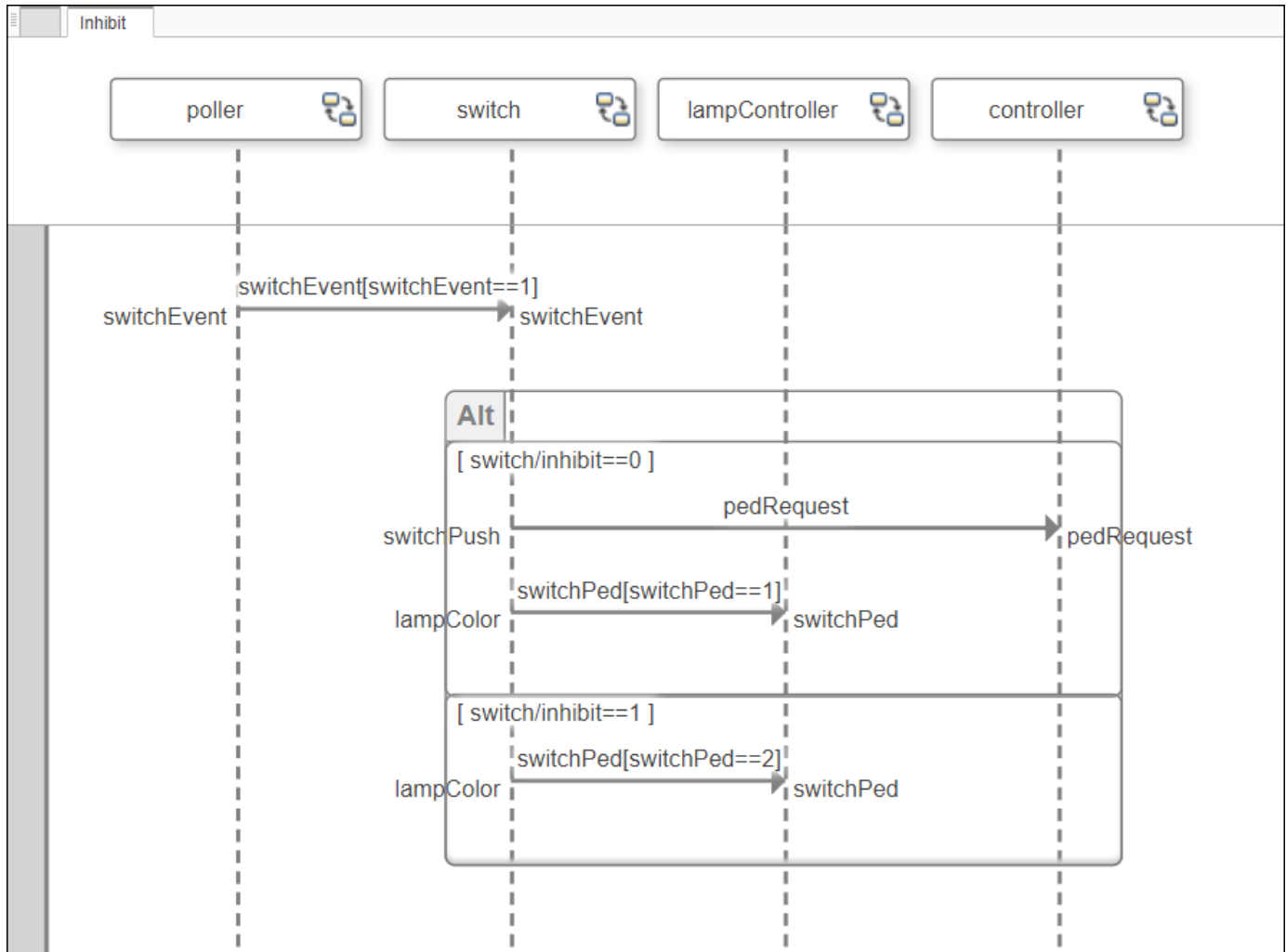


3. PedestrianCross sequence diagram: First, the traffic value is 3, which indicates that the traffic light color is green. The traffic light loops from yellow (2) to red (1) to green (3) and again. When the pedestrian crossing button is pressed and the controller lifeline recognizes a valid

`pedRequest` message, the traffic lamp changes from yellow (2) to red (1), which allows the pedestrians to cross. Then, the main loop continues.



4. Inhibit sequence diagram: The `inhibit` flag determines whether a pedestrian crossing button is set up for pedestrians to press to control the traffic lamp signal on an intersection and cross. When `inhibit` is set to 0, the crossing button exists. When `inhibit` is set to 1, the crossing button does not exist. The `switchEvent` value is 1, which indicates that the pedestrians would like to cross. Once the `switchEvent` value is set to 1, if `inhibit` is 0, the controller lifeline recognizes the `pedRequest` message to initiate a change in the pedestrian lamp color. Additionally, the `switchPed` value is 1, so the traffic lamp will change from yellow to red. Otherwise, if `inhibit` is 1, the `switchPed` value is 2, so the traffic lamp will continue normal operation and not change to red to specifically allow the pedestrians to cross.



Simulate Architecture Model

You can execute the model after setting these variables.

```
createWorkspaceVar("SwitchInputs",[0 11 18],[-1 1 -1]);  
createWorkspaceVar("inhibitFlag",1,0);
```

See Also

More About

- “Author Sequence Diagram for Traffic Light Example” on page 5-38
- “Implement Sequence Diagram Fragments” on page 5-65
- “Use Sequence Diagrams with Architecture Models” on page 5-51
- “Compose Architecture Visually” on page 1-2
- “Describe Component Behavior Using Simulink” on page 5-2
- “Describe Component Behavior Using Stateflow Charts” on page 5-23

Author Sequence Diagram for Traffic Light Example

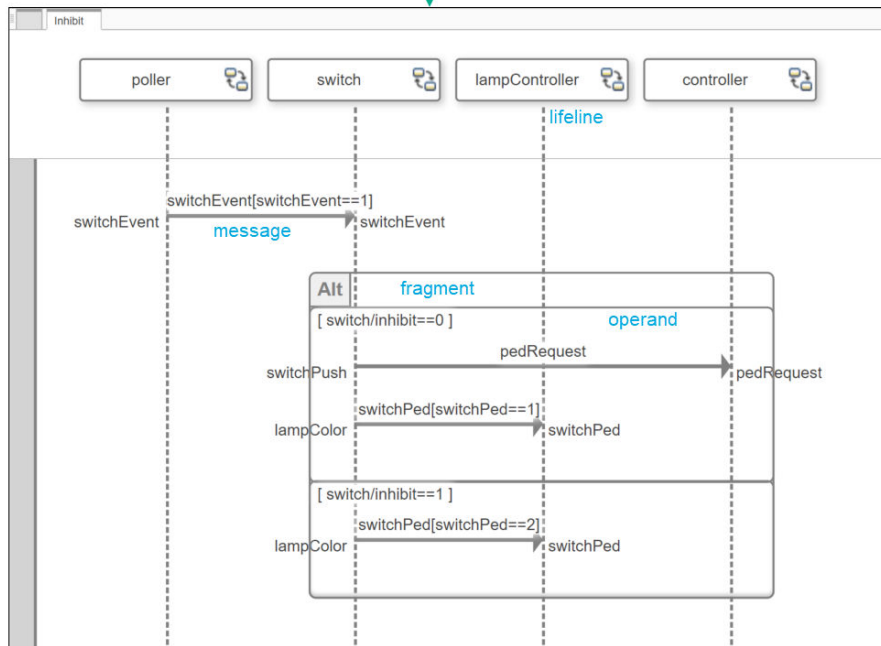
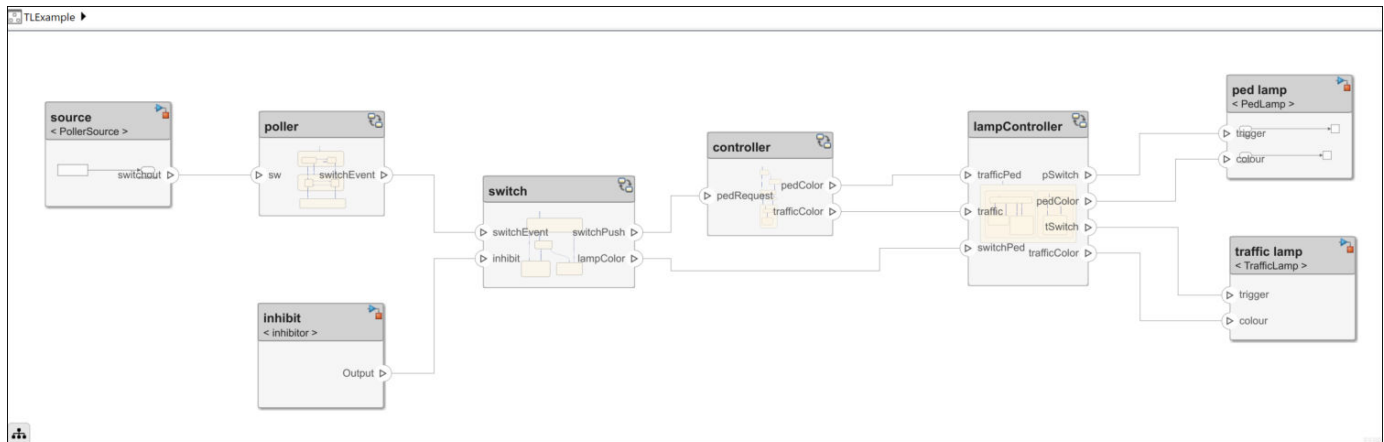
You can create, edit, and simulate sequence diagrams in System Composer by accessing the **Architecture Views Gallery**. You will learn about the basic terminology and functions of a sequence diagram in two stages.

- Add lifelines and messages with message labels including triggers and constraints to represent interactions.
- Include fragments and operands with constraints to further specify the behavior of the interaction.

A lifeline in a sequence diagram represents a component in the architecture. A message represents a communication across a path between the source lifeline and destination lifeline. The path for a message must consist of at least two ports and one connector from the architecture model. With nested messages, the path is more complex due to the hierarchy to be navigated.

For a roadmap of the sequence diagram topics, see “Describe System Behavior Using Sequence Diagrams” on page 5-32.

This figure shows a traffic light architecture model and a corresponding sequence diagram that describes one operative scenario. The traffic light model describes a cycling traffic light, the pedestrian crossing button being pressed, and the lights changing so pedestrians can cross.



Note This example uses Stateflow blocks. If you do not have a Stateflow license, you can open and simulate the model but can only make basic changes, such as modifying block parameters.

Traffic Light Example

This traffic light example contains sequence diagrams to describe pedestrians crossing an intersection. The model describes these steps:

- 1 The traffic signal cycles from red to yellow to green.
- 2 When the pedestrian crossing button is pressed, if the traffic signal is green, the traffic signal transitions from yellow to red for a limited time.

- 3 The pedestrians cross while the walk signal is active.

Open the System Composer model that contains the sequence diagrams.

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

Open the Architecture Views Gallery to view the sequence diagrams.

```
openViews(model)
```

You can execute the model after setting these variables.

```
createWorkspaceVar("SwitchInputs",[0 11 18],[-1 1 -1]);  
createWorkspaceVar("inhibitFlag",1,0);
```

Add Lifelines and Messages

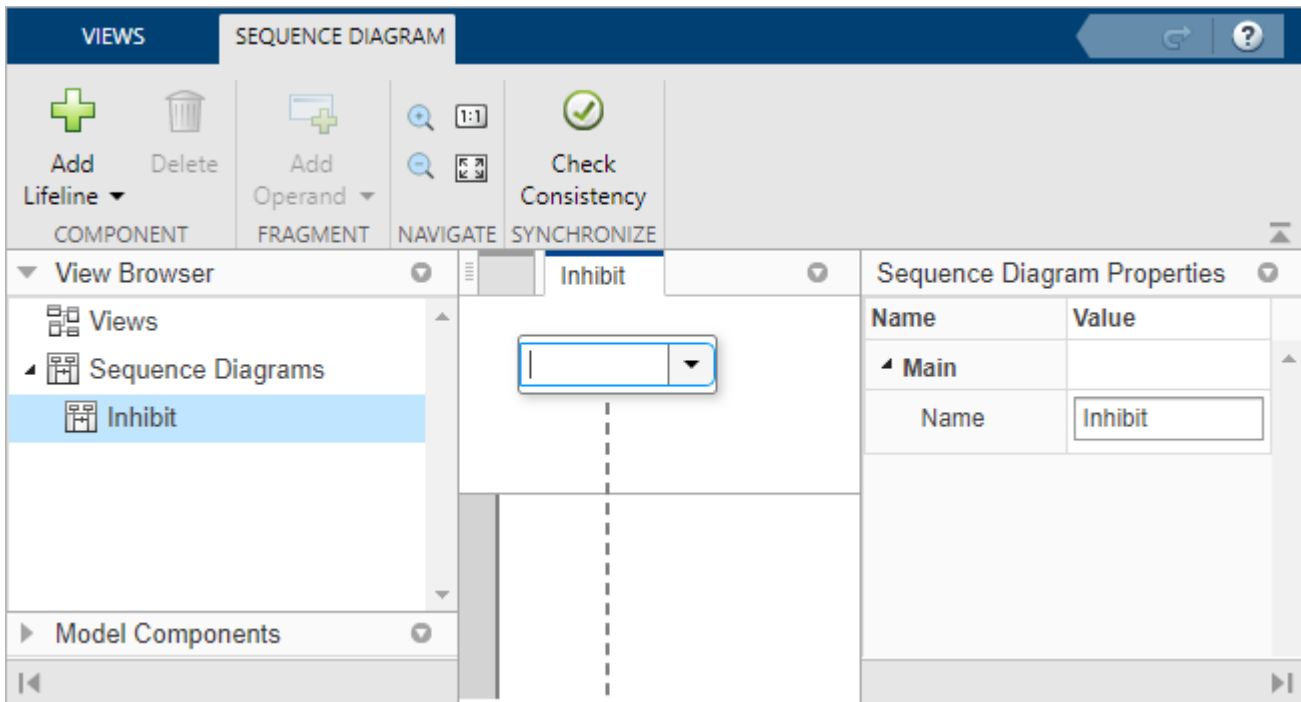
A lifeline is represented by a head and a timeline that proceeds down a vertical dotted line.

The head of a lifeline represents a component in an architecture model.

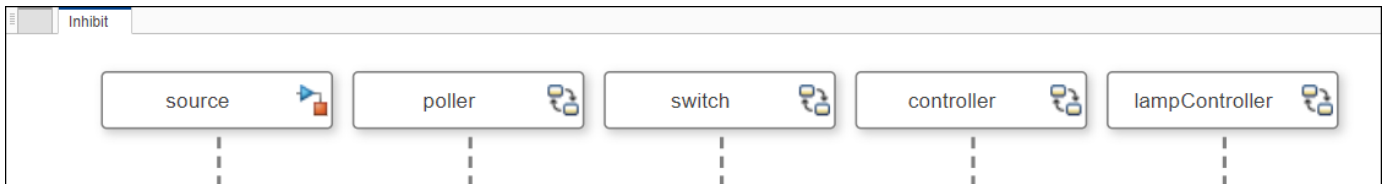
A message includes a send event and a receive event. You can draw messages from one lifeline to another on the vertical dotted lines in a sequence diagram. Messages are clarified with a message label.

A message label has a trigger and a constraint. A trigger determines whether the message occurs. A constraint determines whether the message is valid.

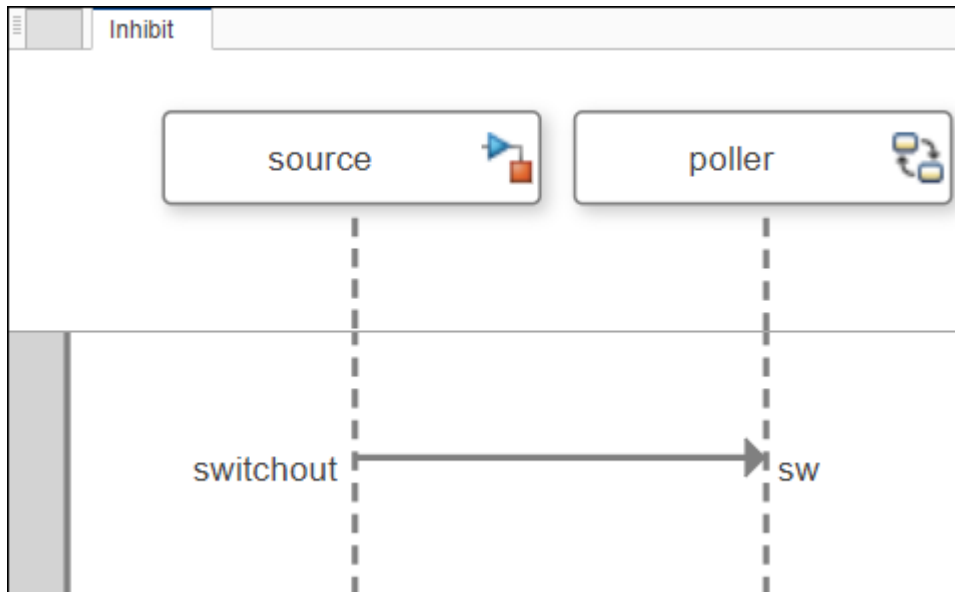
- 1 Navigate to **Modeling > Sequence Diagram** to open sequence diagrams in the **Architecture Views Gallery**.
- 2 To create a new sequence diagram, click **New > Sequence Diagram**.
- 3 A new sequence diagram called SequenceDiagram1 is created in the View Browser, and the **Sequence Diagram** tab becomes active. Under **Element Properties**, rename the sequence diagram Inhibit.
- 4 Select **Component > Add Lifeline** to add a lifeline. A new lifeline with no name is created and is indicated by a dotted line.



- 5 Click the down arrow and select source. The source lifeline detects when the pedestrian presses the crossing button. Add four more lifelines using the down arrow named poller, switch, controller, and lampController. The poller lifeline checks if the pedestrian crossing button has been pressed, switch processes the signal, controller determines which color the pedestrian lamp and traffic light should display, and lampController changes the traffic light colors.



- 6 Draw a line from the source lifeline to the poller lifeline. Start to type sw in the **To** box, which will automatically fill in as you type. Once the text has filled in, select sw.



Since the `switchout` port and `sw` port are connected in the model, a message is created from the `switchout` port to the `sw` port in the sequence diagram.

- 7 A message label has a trigger and a constraint. A trigger determines whether the message occurs, and a constraint determines whether the message is valid. For signal messages, the trigger is called an edge.

You can enter a condition that specifies a triggering edge with a direction and an expression. You can also optionally add a constraint in square brackets to the message. Constraints consist of a MATLAB Boolean expression acting on the inputs of the destination lifeline.

```
direction(signalPort(+|-)positiveReal) [booleanExpression]
```

There are three directions for edges:

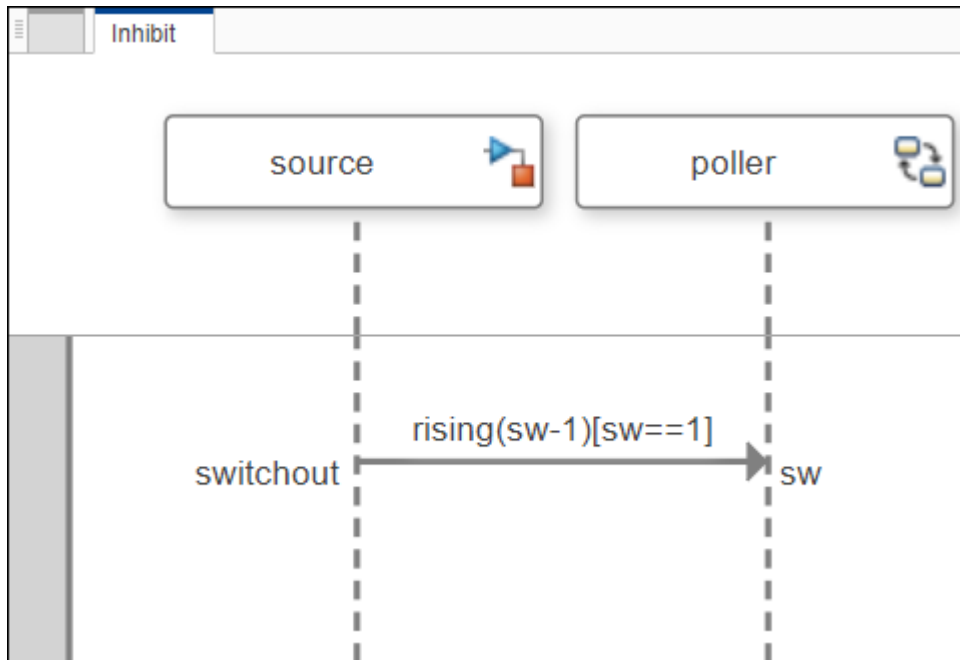
- `crossing` — The edge expression is either rising or falling past zero.
- `rising` — The edge expression is rising from strictly below zero to a value equal to or greater than zero.
- `falling` — The edge expression is falling from strictly above zero to a value equal to or less than zero.

Click on the message and double-click on the empty message label that appears. Enter this condition and constraint.

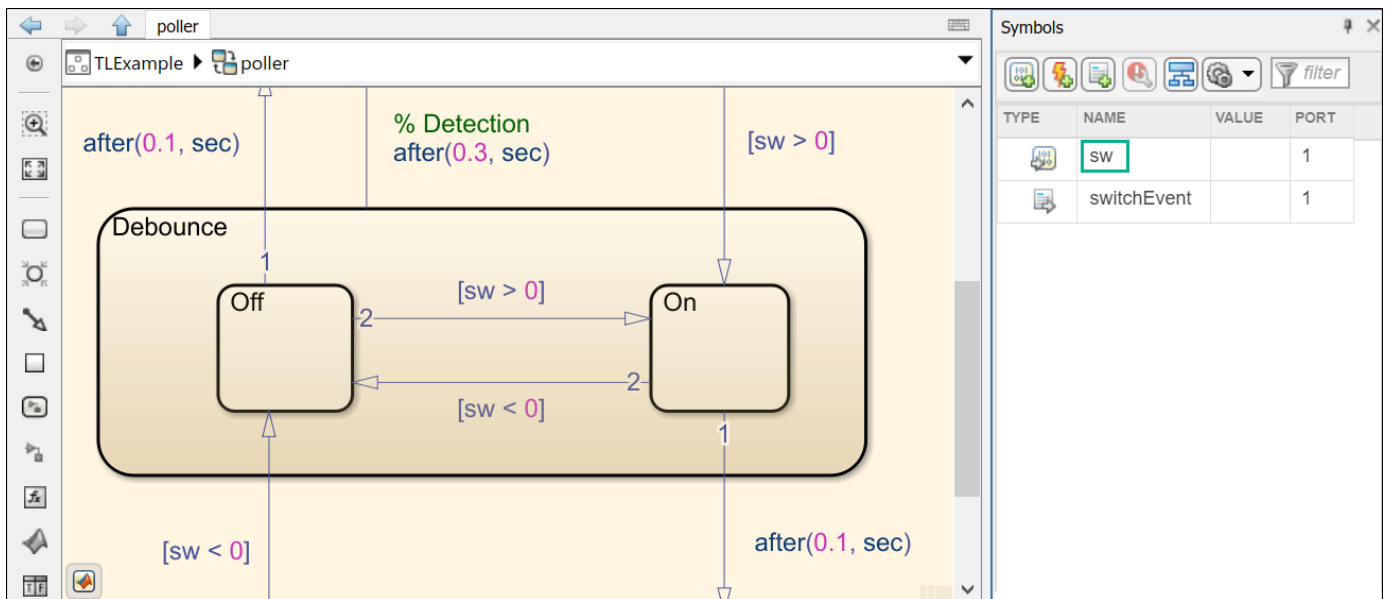
```
rising(sw-1)[sw==1]
```

The message will be triggered when the `sw` signal rises from below 1 to a value of 1 or above. The constraint in square brackets indicates that if `sw` is not equal to 1, the message is invalid.

Note Only destination elements are supported for message labels. In this example, `switchout` is a source element and cannot be included.



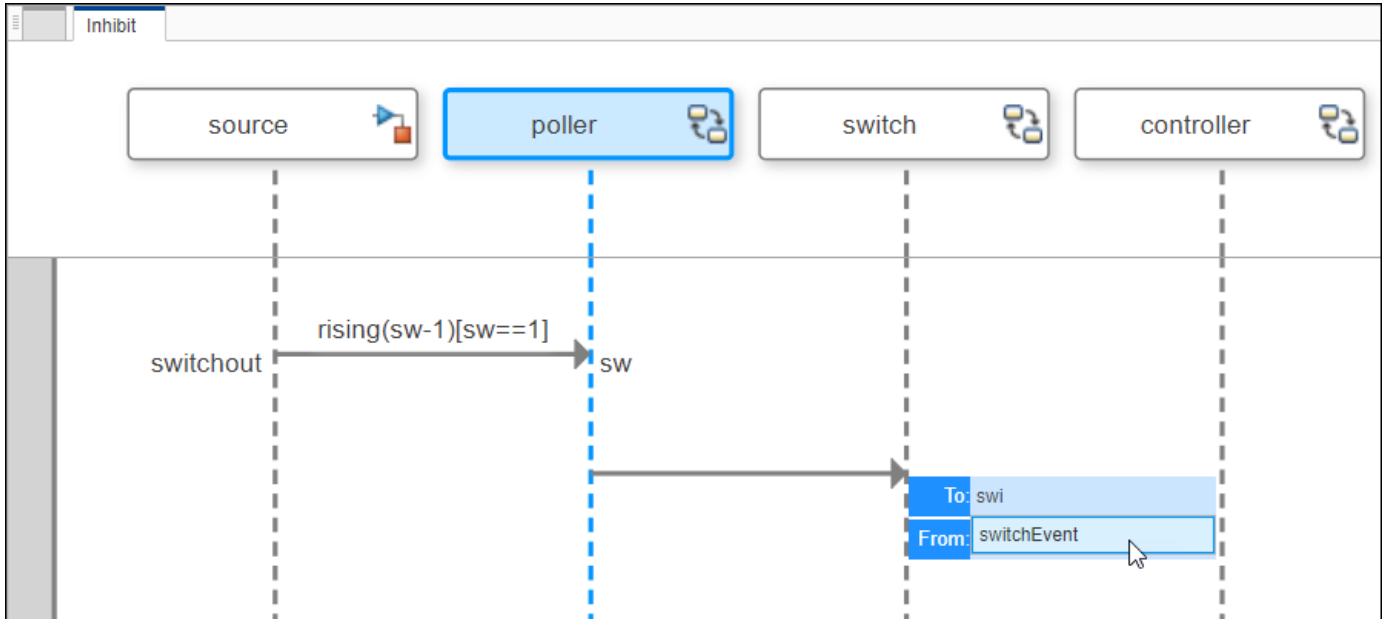
The signal name `sw` is valid input data on the port for a Stateflow chart behavior. The `poller` component with state chart behavior has `sw` in the **Symbols** pane.



Note The signal name can also be a data element on a data interface on a port. Enter **Tab** to autocomplete the port and data element names. For more information, see “Represent System Interaction Using Sequence Diagrams”.

In this example, when the `sw` signal becomes 1, the pedestrian crossing button has been pressed, and a message to the `poller` lifeline is recognized.

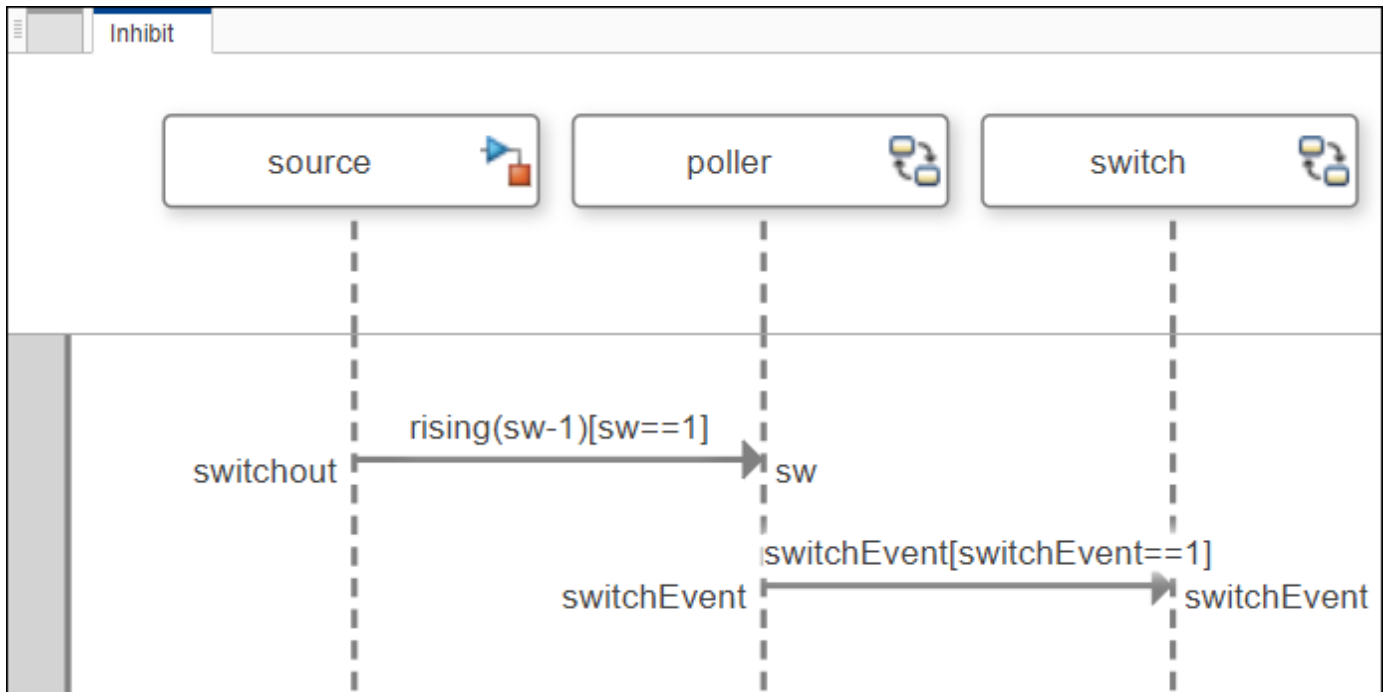
- 8 In addition to signal events, sequence diagrams also support message events. Create a message by drawing a line from the poller lifeline to the switch lifeline. Start typing switchEvent in the **To** box until switchEvent is available to select.



Since there is an existing connection in the architecture model, a message is created from source port switchEvent.

- 9 Click the message and double-click the empty message label that appears. Enter this condition representing the port and constraint.

`switchEvent[switchEvent==1]`



When the message `switchEvent` is received and its value is 1, the message has occurred and is valid.

Add Fragments and Operands

A fragment indicates how a group of messages within are executed or interact in a sequence diagram.

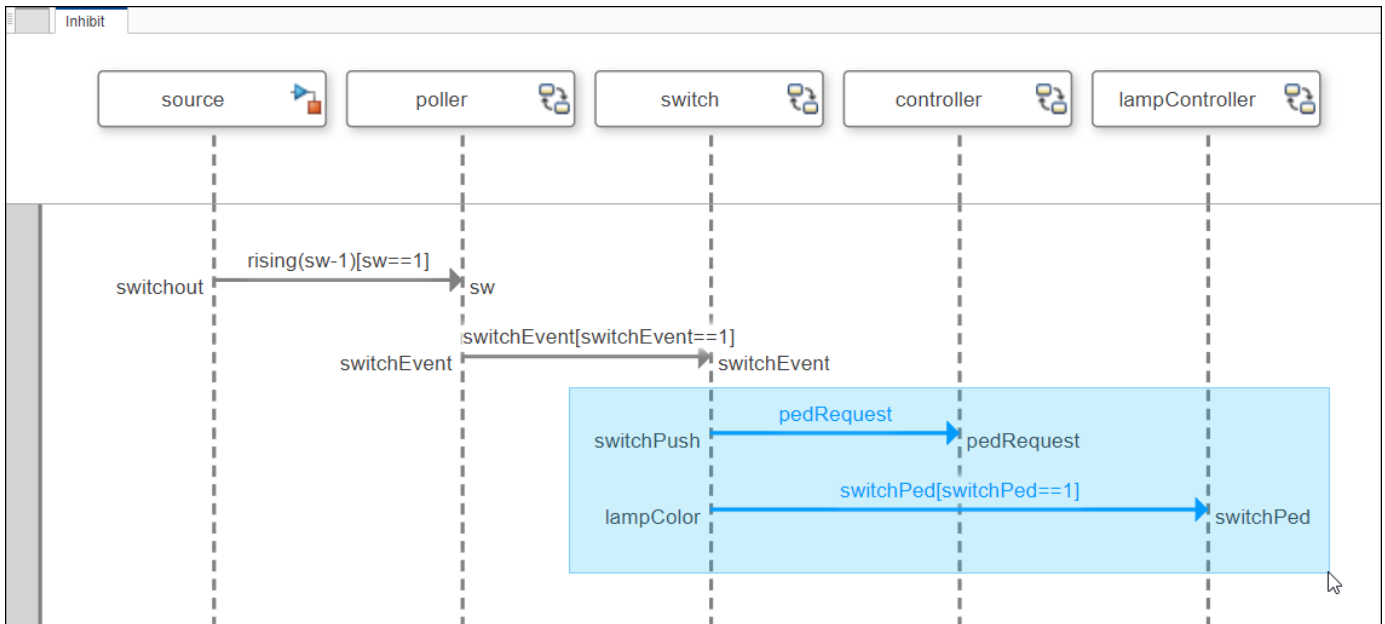
A fragment is used to model complex sequences, such as alternatives, in a message-based behavior diagram.

An operand is a region in a fragment. Fragments have one or more operands depending on the kind of fragment. Operands can contain messages and additional fragments.

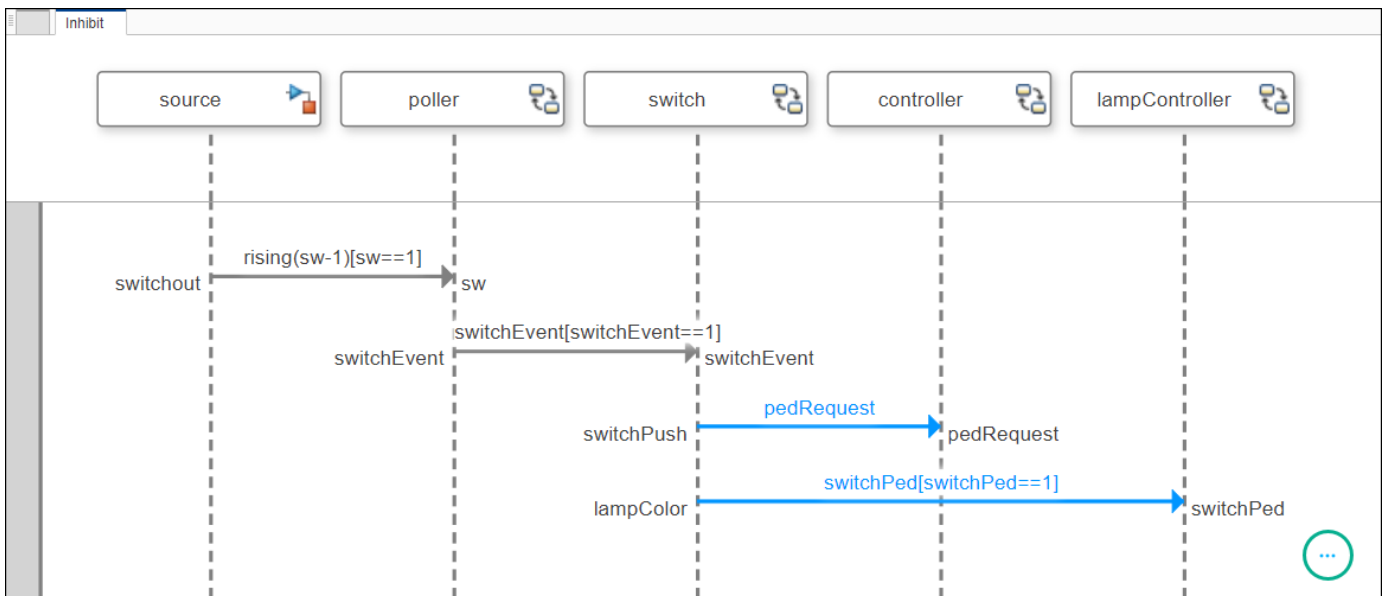
Each operand can include a constraint to specify whether the messages inside the operand execute. You can express the precondition of an operand as a MATLAB Boolean expression using the inputs of any lifeline.

To access the menu of fragments:

- 1 Click and drag to select two messages.



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

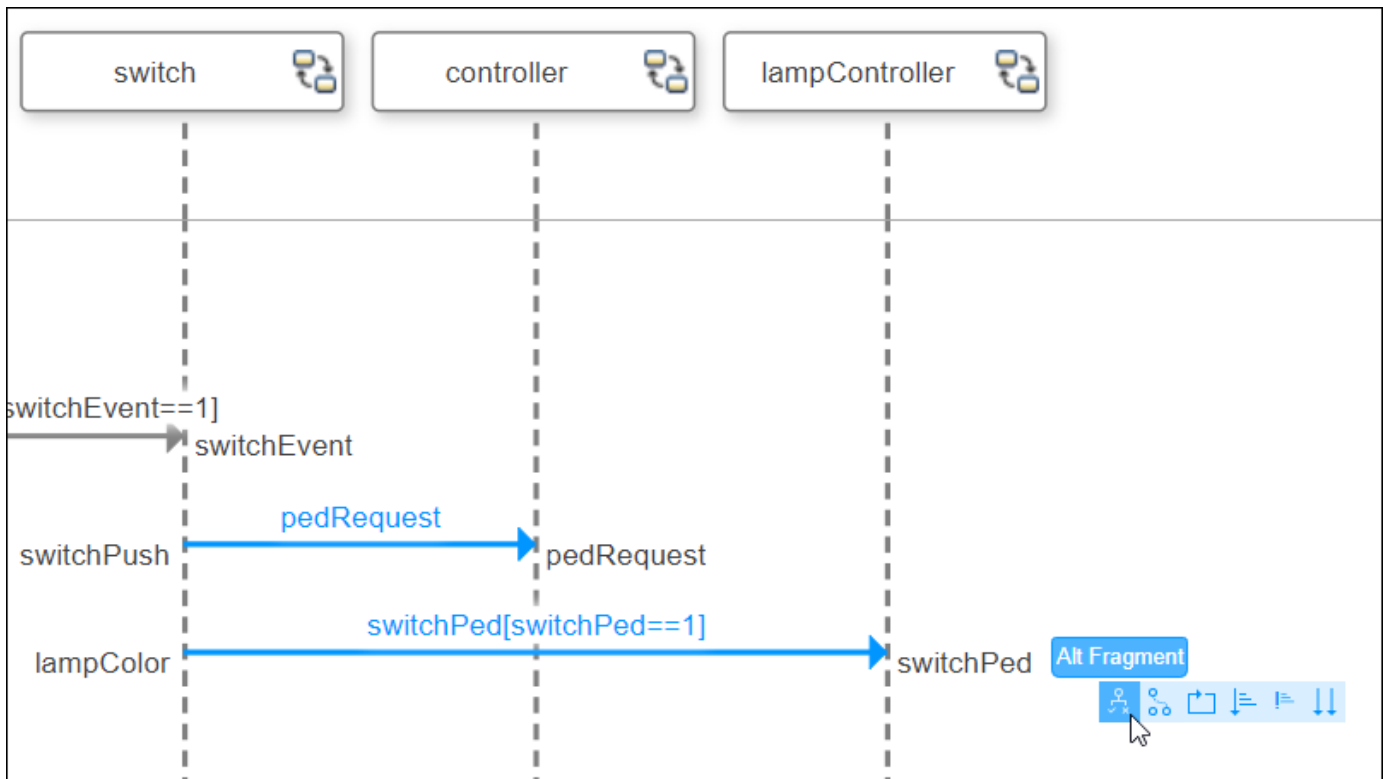


3 A list of fragments appears:

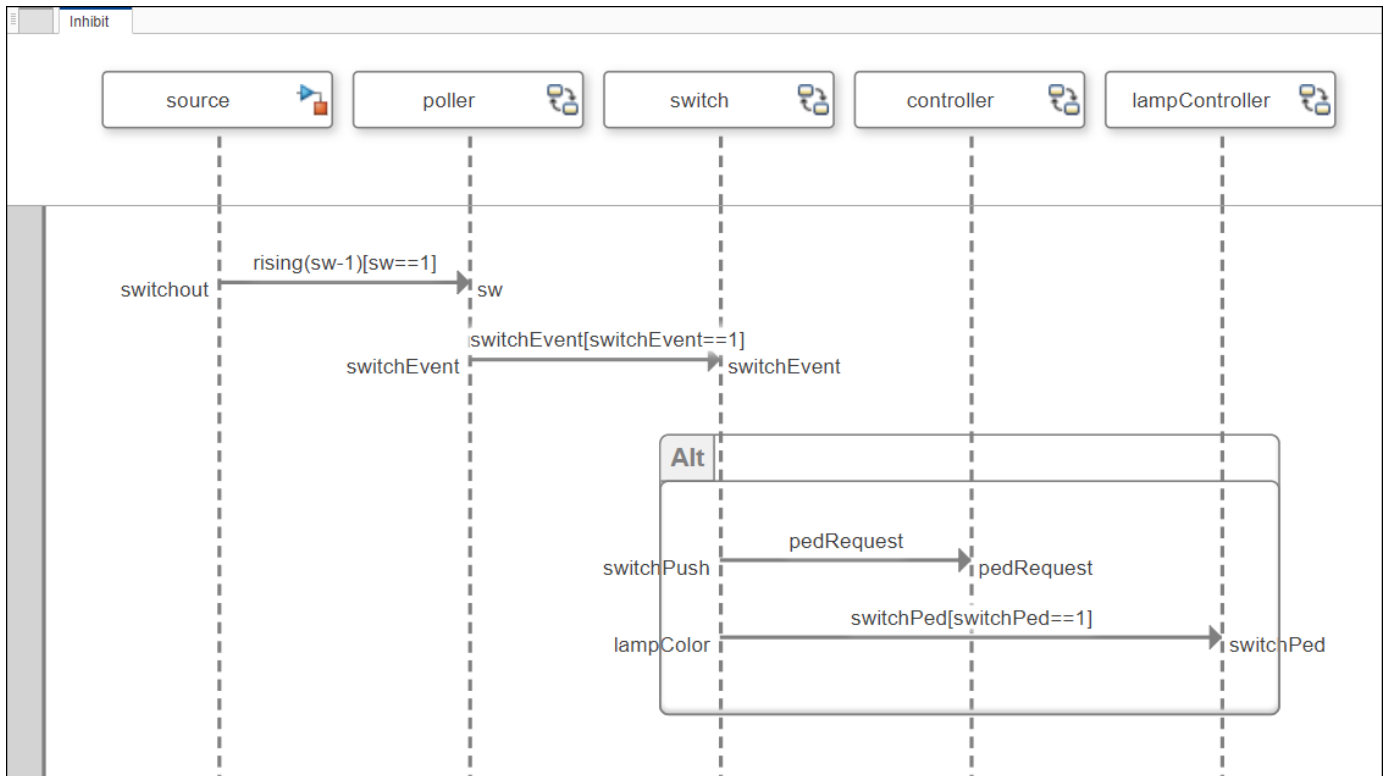
- Alternative (Alt) fragment
- Optional (Opt) fragment
- Loop (Loop) fragment
- Weak sequencing (Seq) fragment
- Strict sequencing (Strict) fragment
- Parallel (Par) fragment

For more information, see “Implement Sequence Diagram Fragments” on page 5-65.

Select the "Alt Fragment" on page 5-71 fragment.



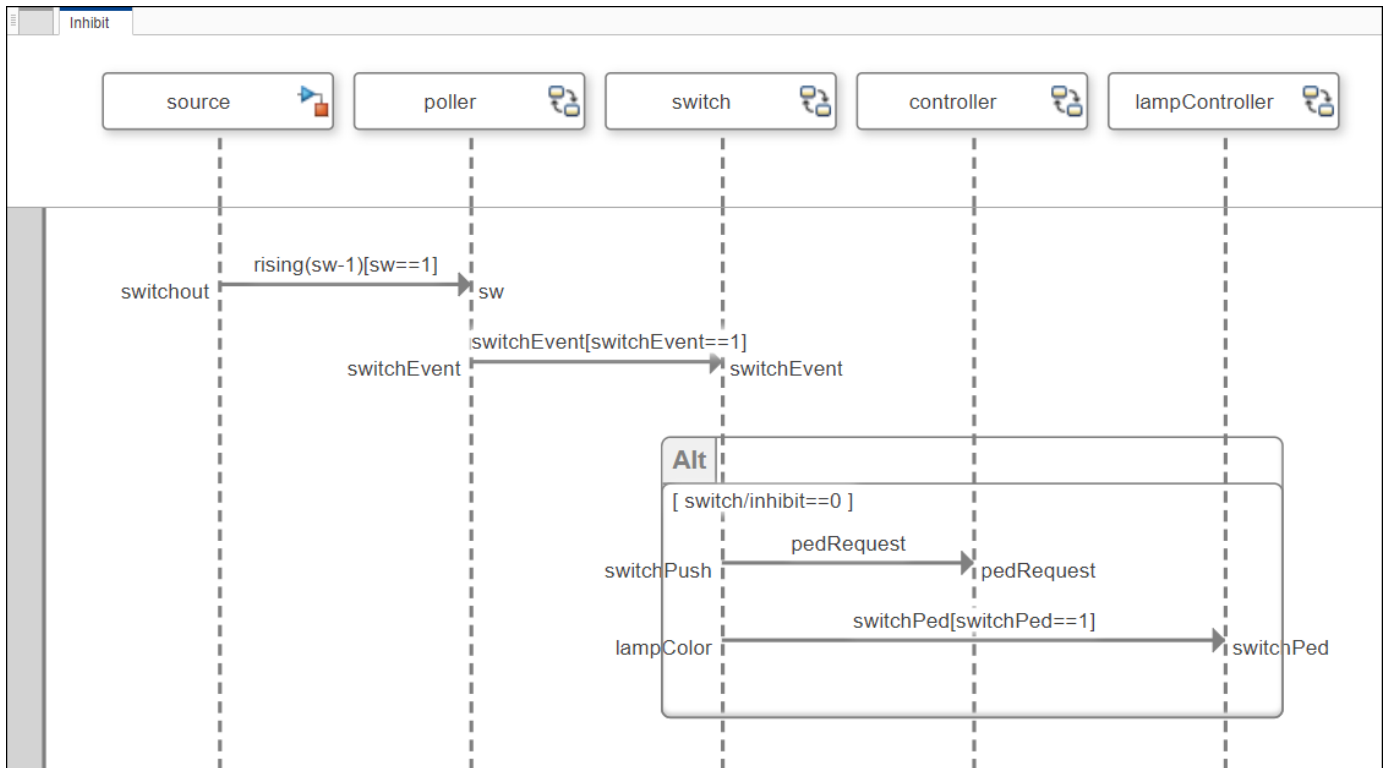
- 4 The "Alt Fragment" on page 5-71 fragment is added to the sequence diagram with a single operand that contains the selected messages.



- 5 Select the fragment to enter an operand condition. Choose a fully qualified name for input data and use a constraint condition relation.

switch/inhibit==0

This constraint is a precondition that determines when the operand is active. This constraint specifies that the `inhibit` flag is set to 0. Thus, pedestrian crossing is allowed at this intersection using a pedestrian lamp.



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

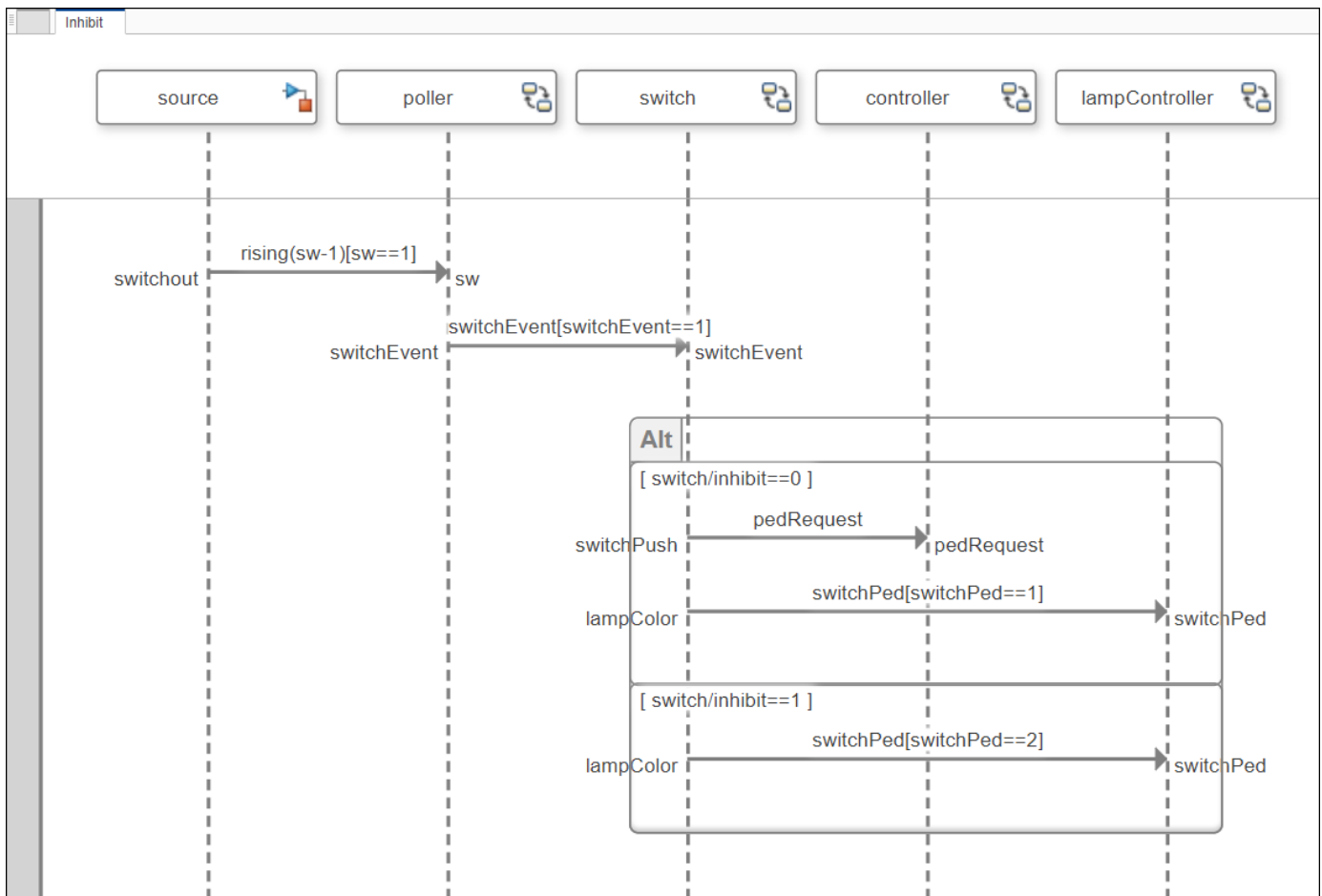
- Highlight the first operand under the “Alt Fragment” on page 5-71 fragment and select from the toolstrip **Fragment > Add Operand > Insert After**. A second operand is added.

Add a constraint condition relation to the second operand. The second operand in an “Alt Fragment” on page 5-71 fragment represents an **elseif** condition for which the message will be executed.

`switch/inhibit==1`

This constraint represents when the `inhibit` flag is set to 1. Thus, pedestrian crossing is not controlled by a walk signal on that intersection.

Create a message with a message label inside the second operand.



For the first operand in the “Alt Fragment” on page 5-71 fragment, since the inhibit flag is set to 0, the first message to the controller lifeline is recognized when the pedRequest message is valid. Then, when the switchPed message value is 1, the lampController component behavior allows pedestrians to cross.

For the second operand in the “Alt Fragment” on page 5-71 fragment, since the inhibit flag is set to 1, the switch bypasses the controller. The message switchPed with a value of 2 goes directly to the lampcontroller which does not affect the traffic signal. Pedestrian crossing is not specifically supported in this traffic intersection.

See Also

More About

- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Compose Architecture Visually” on page 1-2
- “Describe Component Behavior Using Simulink” on page 5-2
- “Describe Component Behavior Using Stateflow Charts” on page 5-23
- “Define Port Interfaces Between Components” on page 3-2

Use Sequence Diagrams with Architecture Models

You can author sequence diagrams to describe expected system behavior as a sequence of interactions between components of a System Composer architecture model. Lifelines correspond to components in an architecture model, and messages correspond to the connectors between the components. 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.

For a roadmap of the sequence diagram topics, see “Describe System Behavior Using Sequence Diagrams” on page 5-32.

This traffic light example will show you how to:

- Create a sequence diagram.
- Add child lifelines in a sequence diagram.
- Interact with root architecture ports in a sequence diagram using gates.
- Co-create components and keep the architecture model and the sequence diagram in sync.
- Create messages in a sequence diagram.
- Use the model browser to add components.

Note This example uses Stateflow blocks. If you do not have a Stateflow license, you can open and simulate the model but can only make basic changes, such as modifying block parameters.

Traffic Light Example with Hierarchy for Sequence Diagrams

This traffic light example contains sequence diagrams to describe pedestrians crossing an intersection. The model describes these steps:

- 1 The traffic signal cycles from red to yellow to green.
- 2 When the pedestrian crossing button is pressed, if the traffic signal is green, the traffic signal transitions from yellow to red for a limited time.
- 3 The pedestrians cross while the walk signal is active.

Open the System Composer model that contains the sequence diagrams.

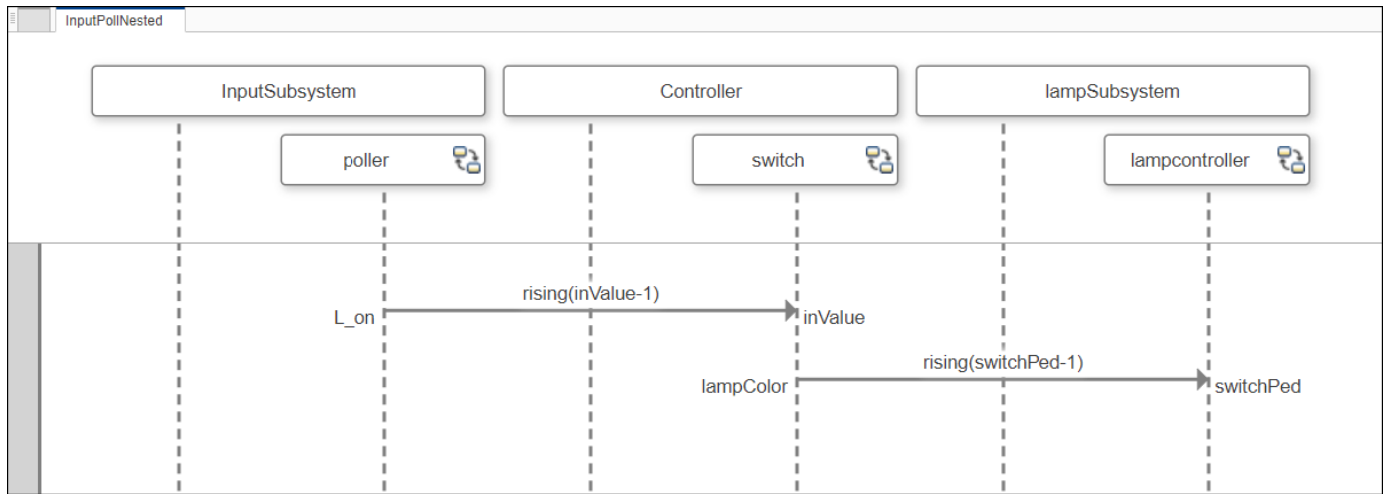
```
model = systemcomposer.openModel('TrafficLight');
```

Open the Architecture Views Gallery to view the sequence diagrams.

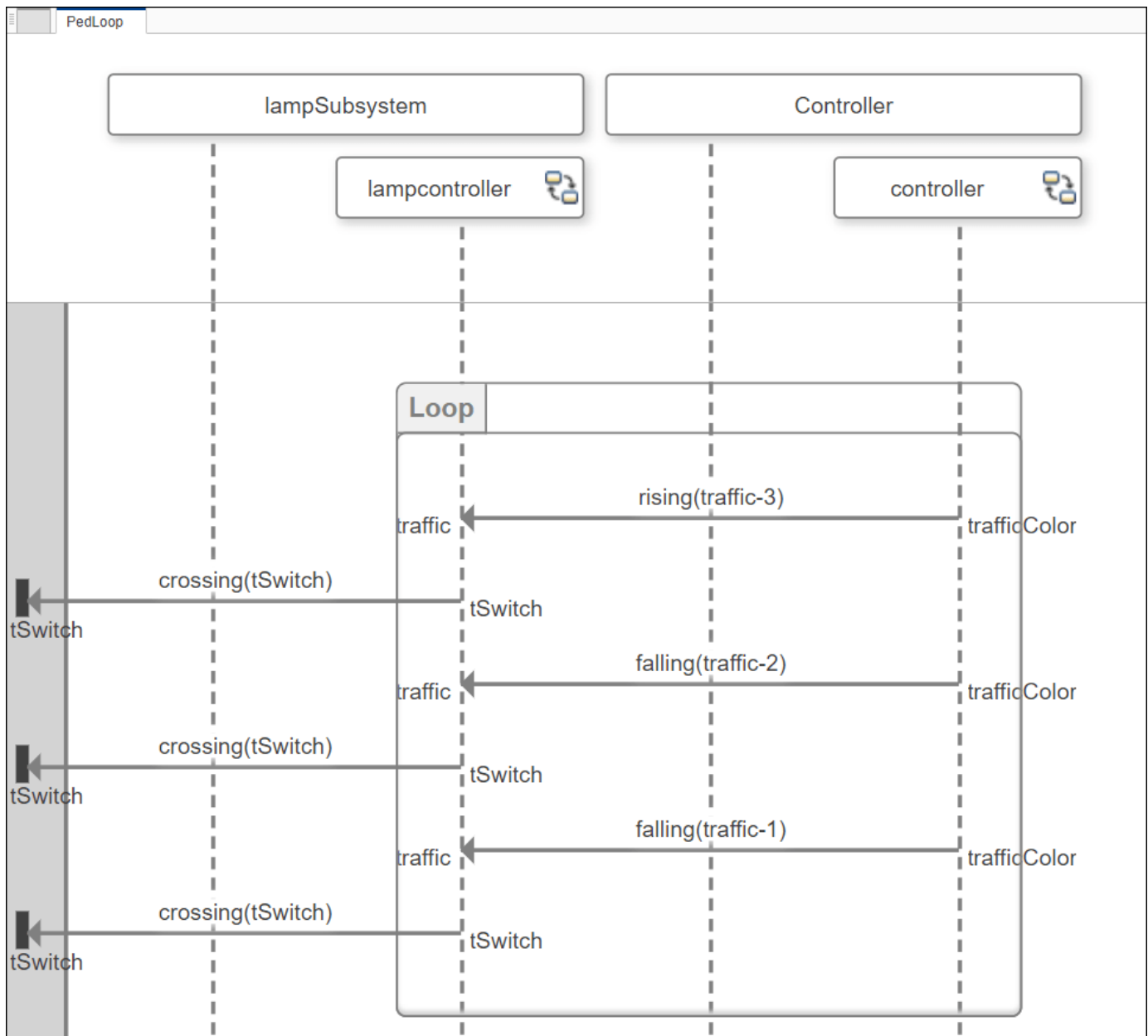
```
openViews(model)
```

The sequence diagrams in this example represent operative scenarios in the architecture model.

1. **InputPollNested** sequence diagram: When the `poller` recognizes a signal event as `inValue` rises to 1, the pedestrian crossing button is pressed. Next, the `switch` lifeline recognizes a signal event to `lampcontroller` as `switchPed` rises to 1, which activates the pedestrian crossing signal.



2. PedLoop sequence diagram: The traffic lamp changes trafficColor from green (3) to yellow (2) to red (1). After each traffic color change, the tSwitch value becomes 0, which indicates that the traffic lamp has been changed. The cycle repeats in a loop for several iterations before the pedestrian crossing button is pressed.



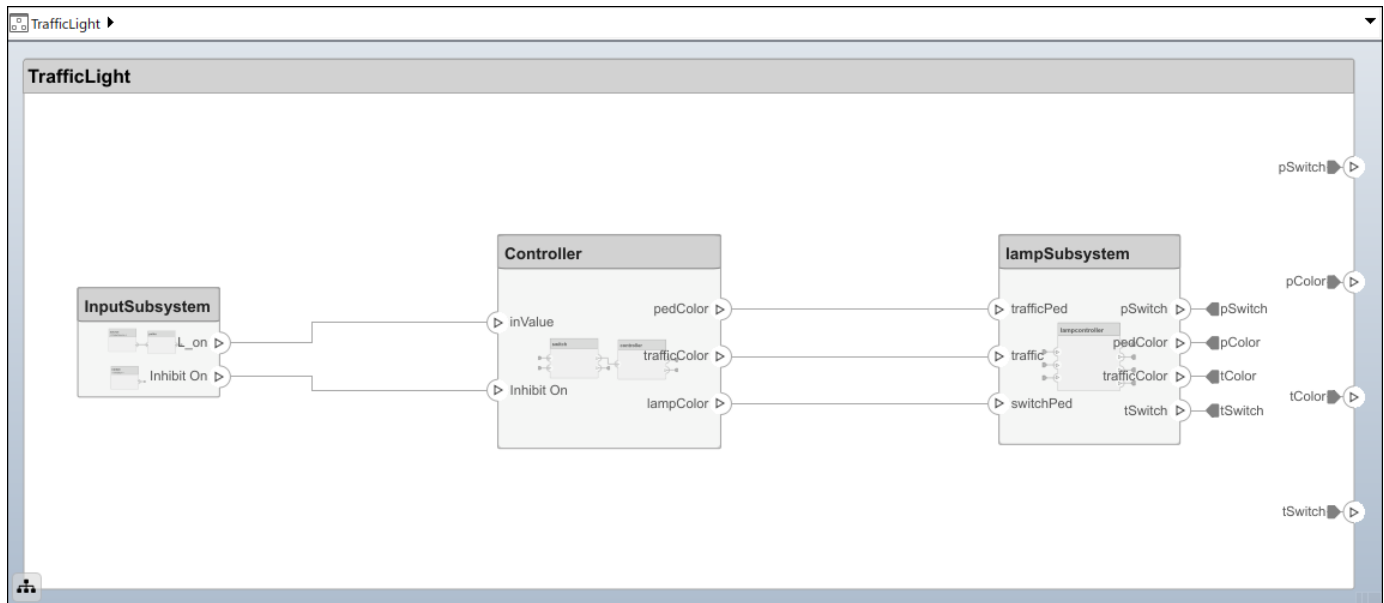
Simulate Architecture Model

You can execute the model after setting these variables.

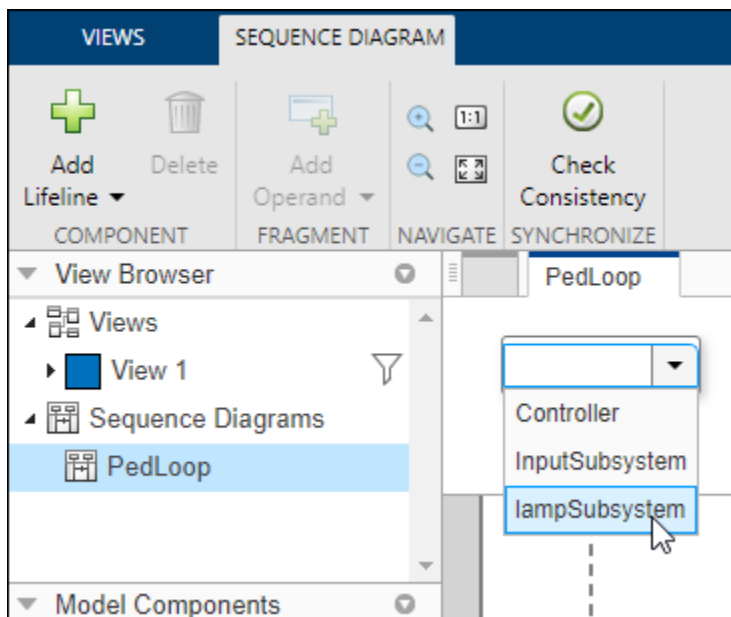
```
createWorkspaceVar("SwitchInputs",[0 11 18],[-1 1 -1]);
createWorkspaceVar("inhibitFlag",1,0);
```

Create Sequence Diagram

Use an architecture model in System Composer to represent a traffic light example.



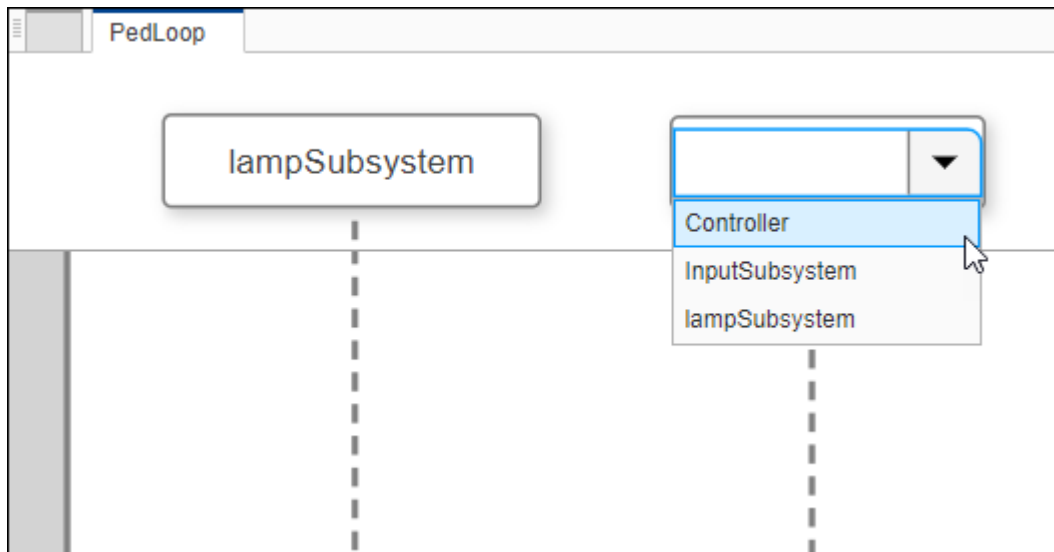
- 1 Navigate to **Modeling > Sequence Diagram** to open sequence diagrams in the **Architecture Views Gallery**.
- 2 To create a new sequence diagram, click **New > Sequence Diagram**.
- 3 In **Element Properties** on the right, enter the name PedLoop.
- 4 Select **Component > Add Lifeline** from the menu. A box with a vertical dotted line appears on the canvas. This is the new lifeline.
- 5 Click the down arrow on the lifeline to view available options. Select the component named lampSubsystem to be represented by the lifeline.



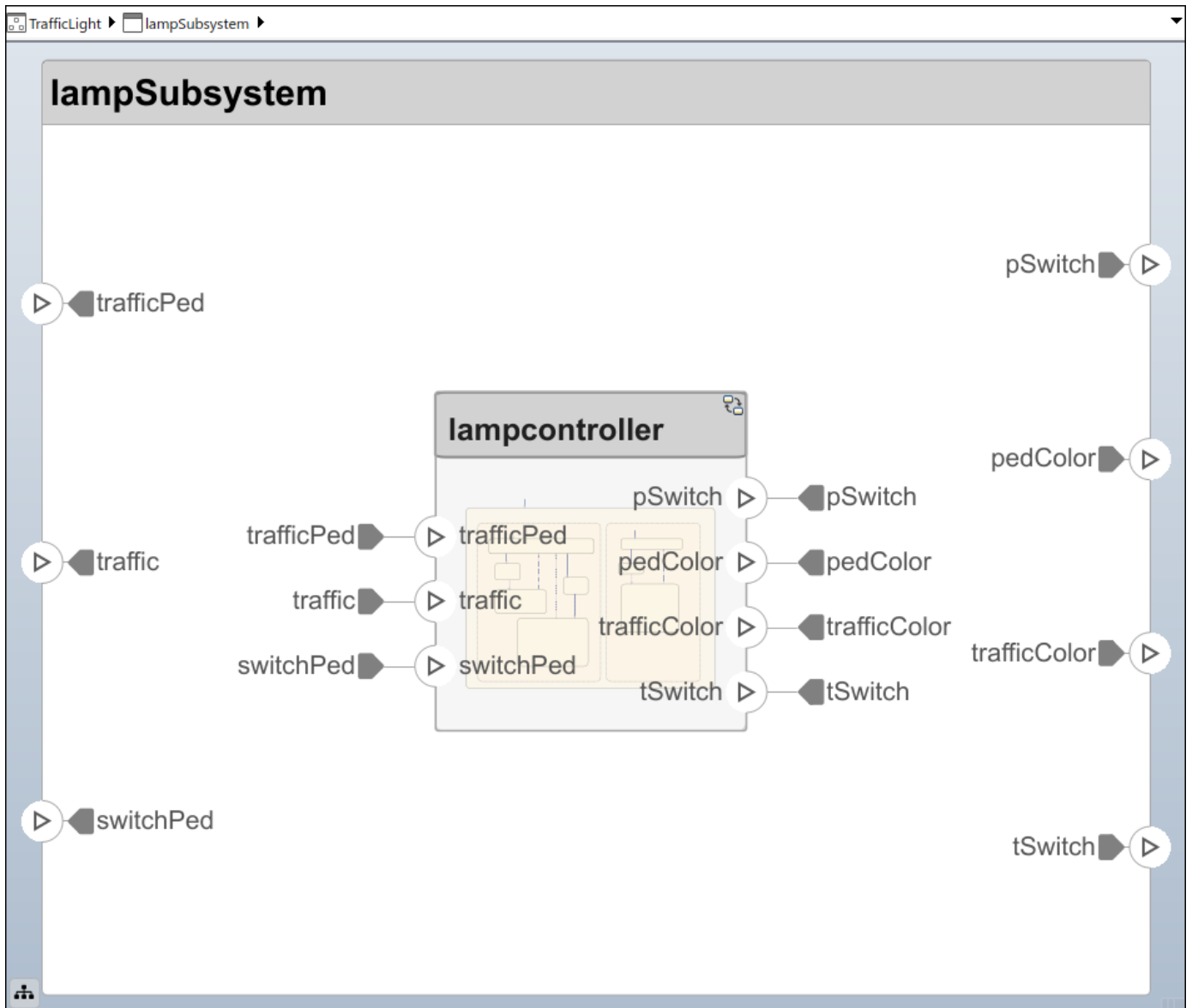
Add Child Lifelines to Sequence Diagram

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

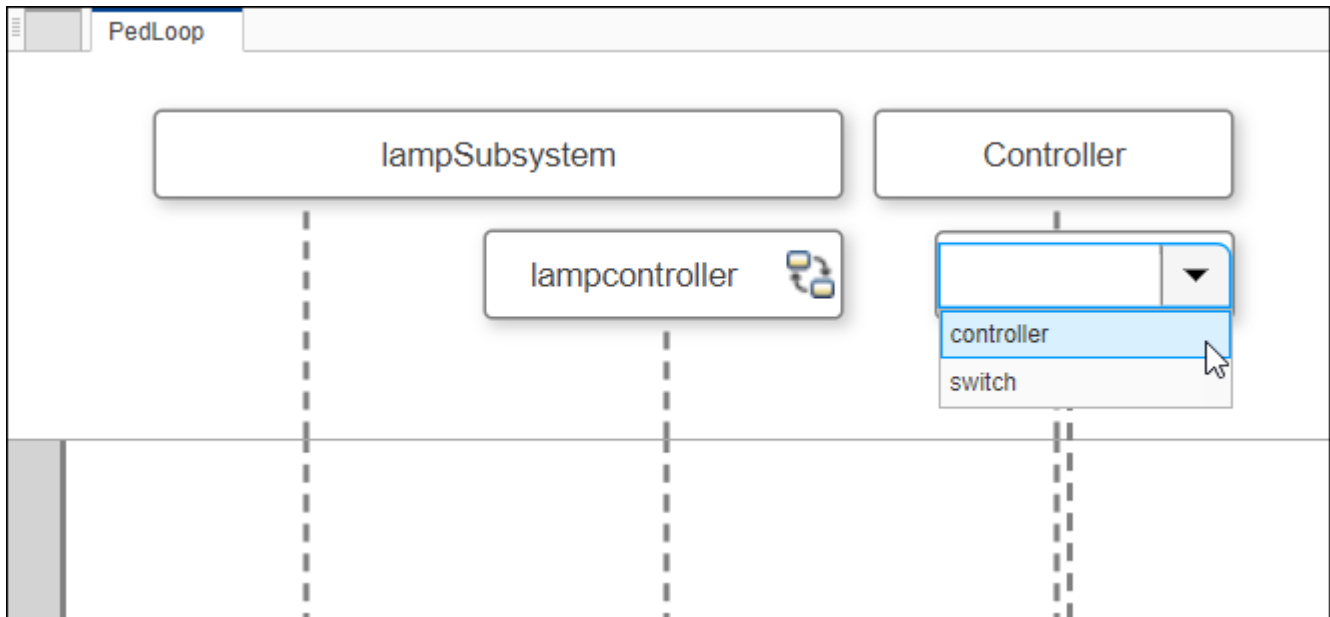
- 1 From the menu, select **Component > Add Lifeline**. From the list that appears, select the Controller component.



- 2 Child components called `lampcontroller` and `controller` are located inside the `lampSubsystem` and `Controller` components, respectively.

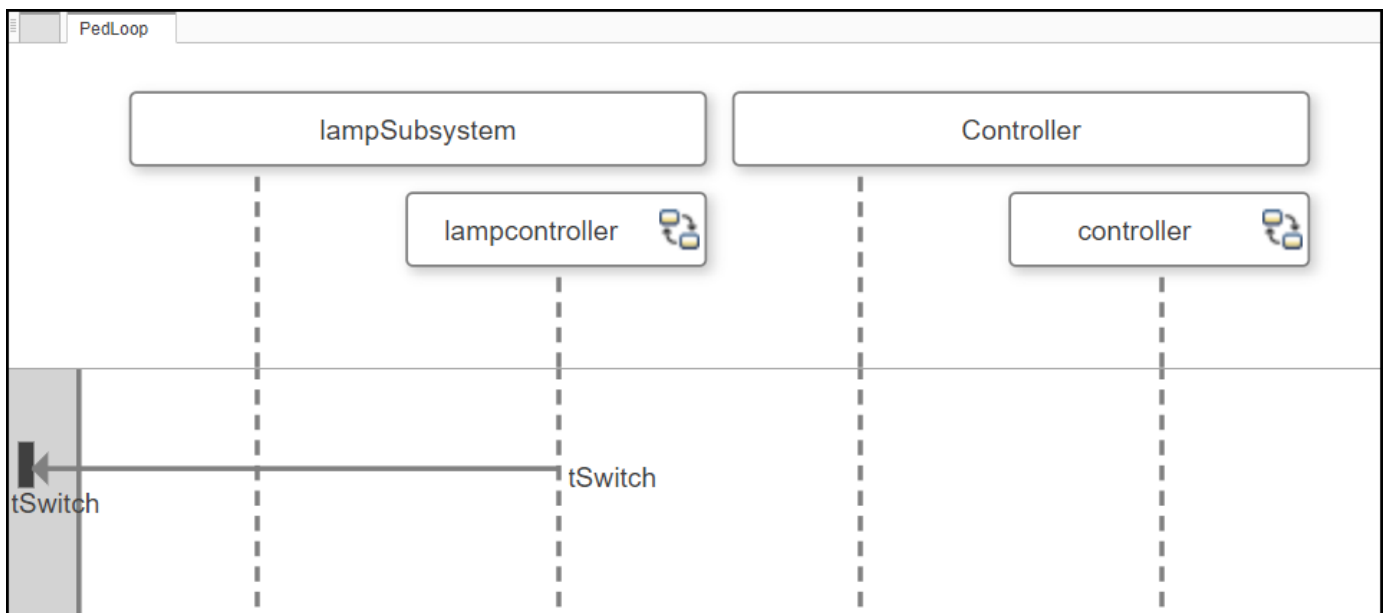


- 3 Select the `lampSubsystem` lifeline. Navigate to **Component > Add Lifeline > Add Child Lifeline**. Select `lampcontroller`. The `lampcontroller` child lifeline is now situated below `lampSubsystem` in the hierarchy.
- 4 Repeat these steps for the `Controller` lifeline to add the controller child lifeline.

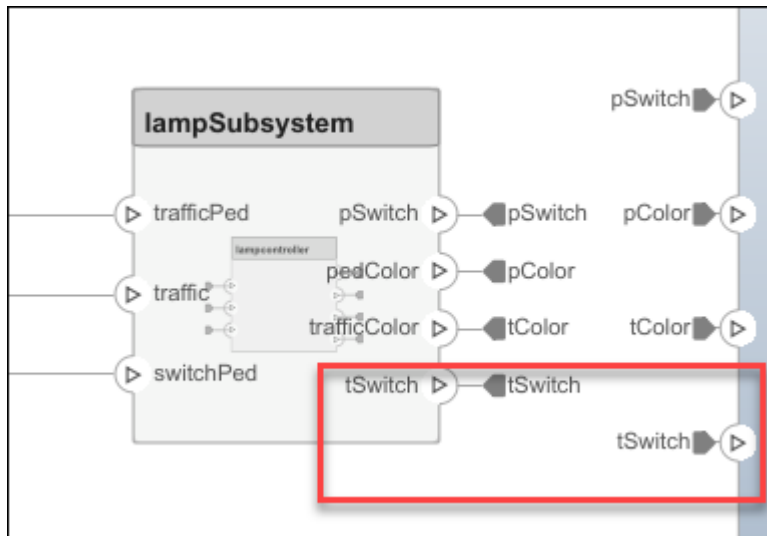


Create Sequence Diagram Gates

- 1 Select the `lampcontroller` lifeline, then click and drag it to the gutter region. Start typing `tSwitch` into the **To** box and select `tSwitch` from the list. See that a gate called `tSwitch` has been created with a message from the `lampcontroller` lifeline at the port `tSwitch`.



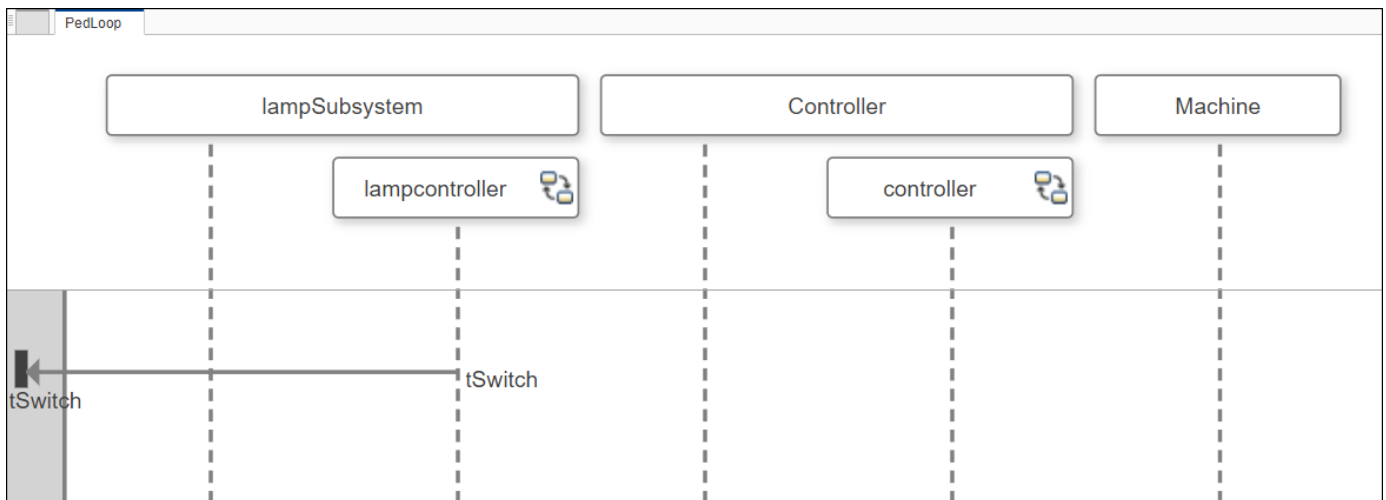
- 2 Return to the architecture diagram. Observe that `tSwitch` is a root architecture port connected to the `lampcontroller` component in the hierarchy through the `lampSubsystem` component.



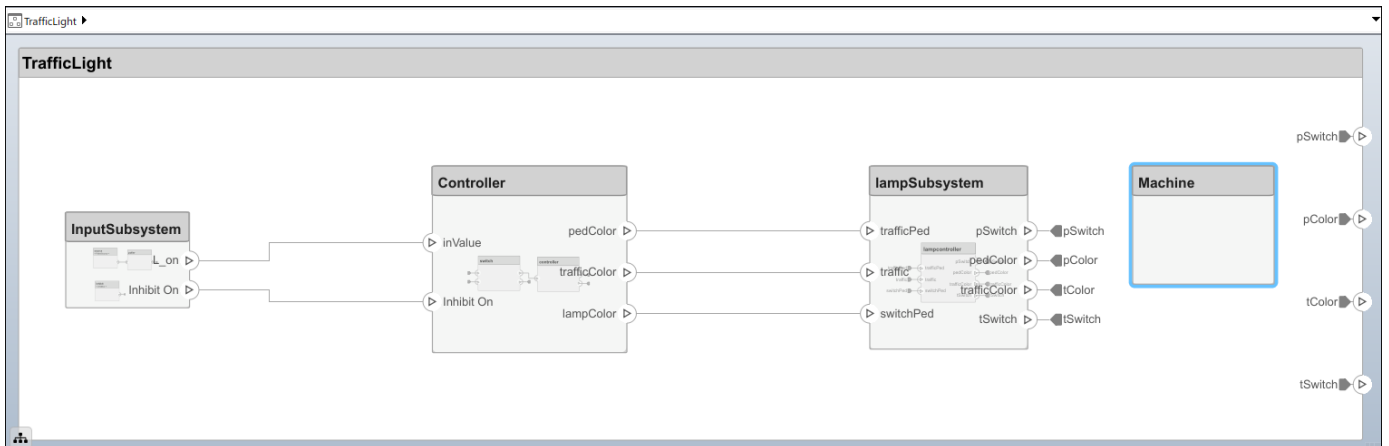
Co-Create Components

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.

- 1 From the toolstrip menu, select **Component > Add Lifeline**. Another box with a vertical dotted line appears on the canvas to represent a lifeline. In the box, enter the name of a new component named Machine.

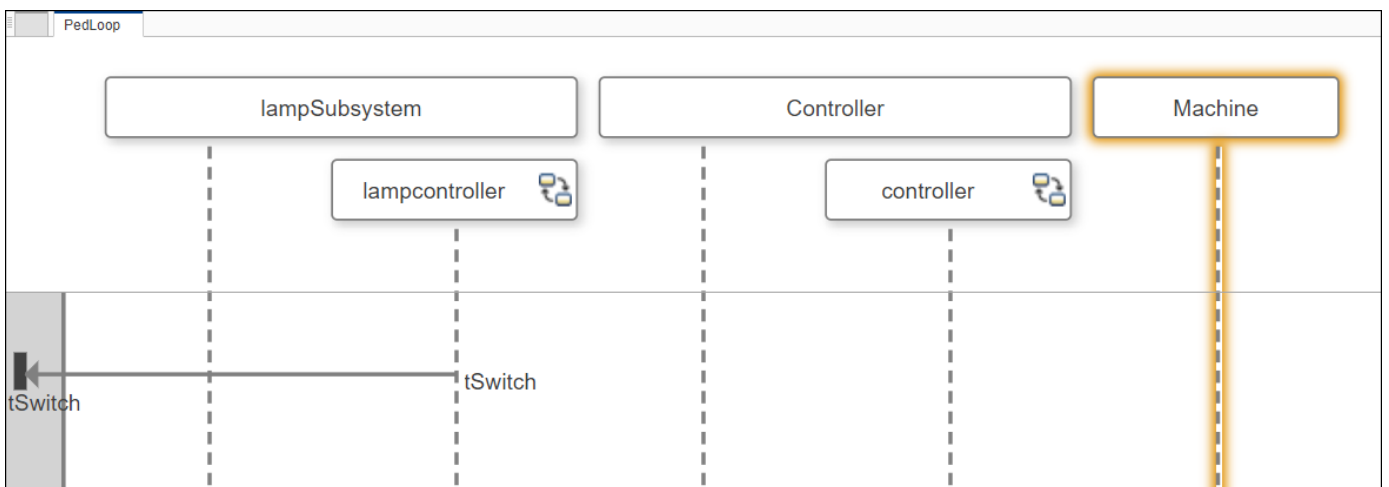


- 2 Observe that the Machine component is co-created in the architecture diagram.



Synchronize Sequence Diagram and Model

- 1 Remove the Machine component from the architecture diagram.
- 2 Return to the sequence diagram and select **Synchronize > Check Consistency**. See that the Machine lifeline is highlighted, as it does not correspond to a component.



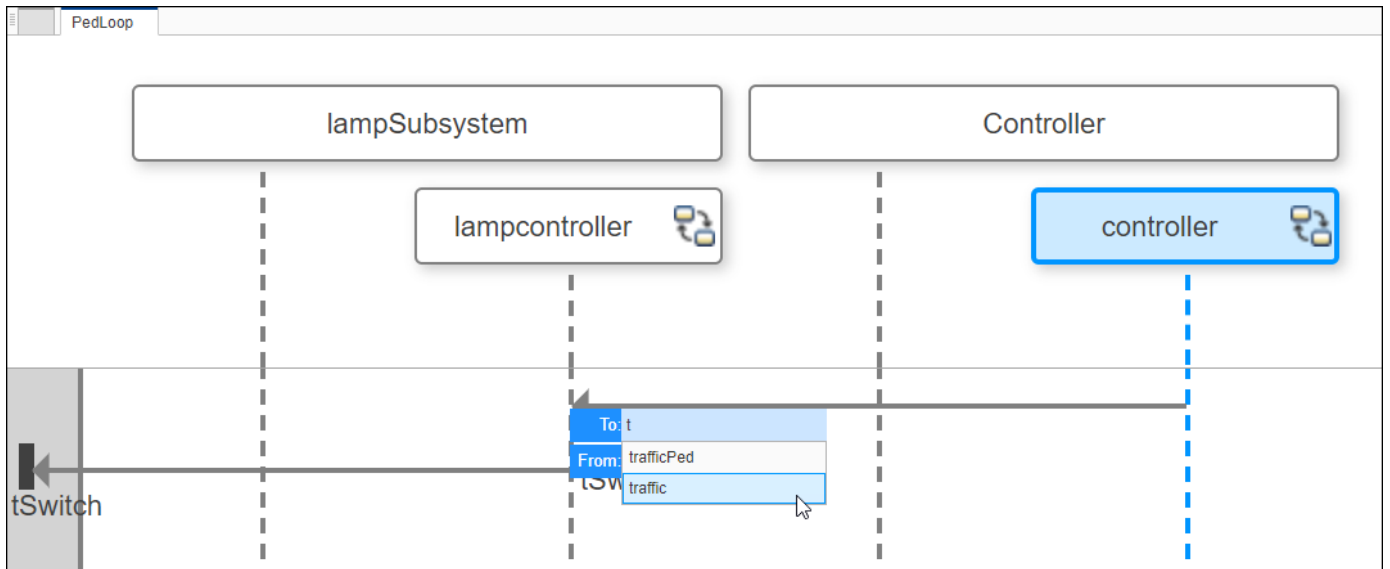
- 3 To restore consistency, either remove the Machine lifeline or click **Undo** in the architecture model to restore the Machine component.
- 4 Click **Check Consistency** again.

For advanced sequence diagram synchronization techniques, see “Synchronize Sequence Diagrams and Architecture Models” on page 5-81.

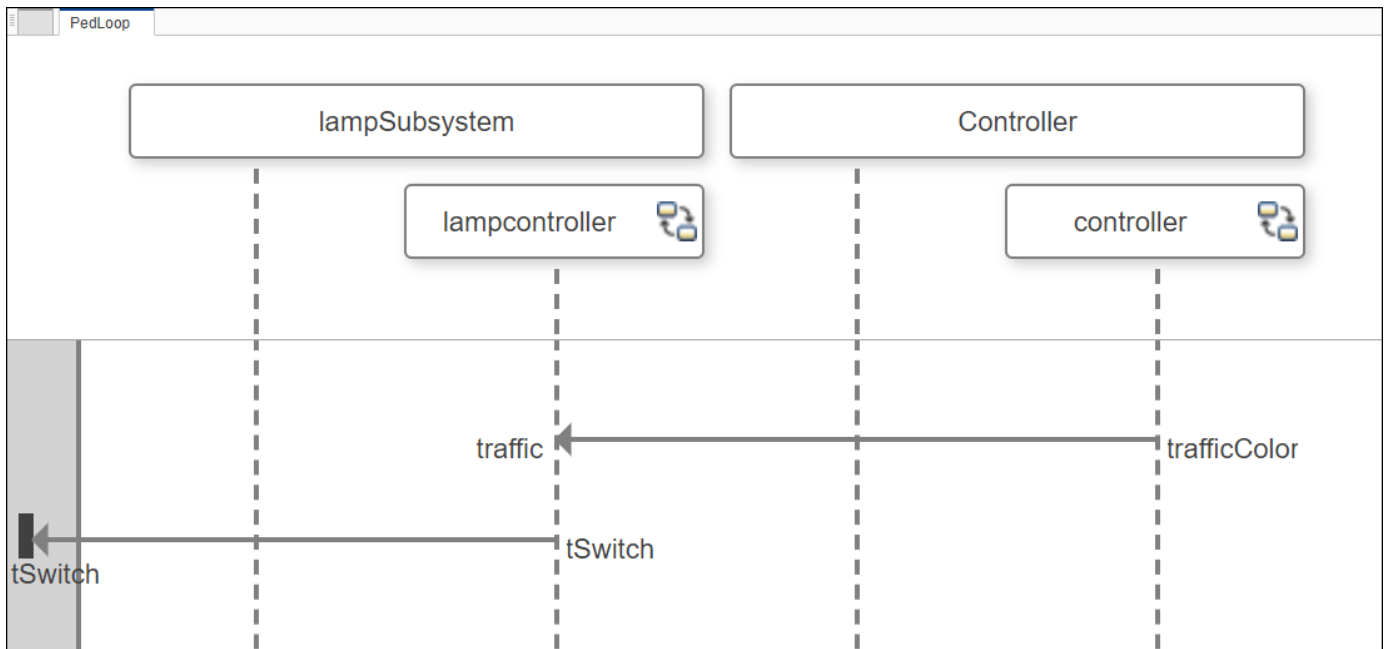
Create Messages in Sequence Diagram

You can create a message from an existing connection.

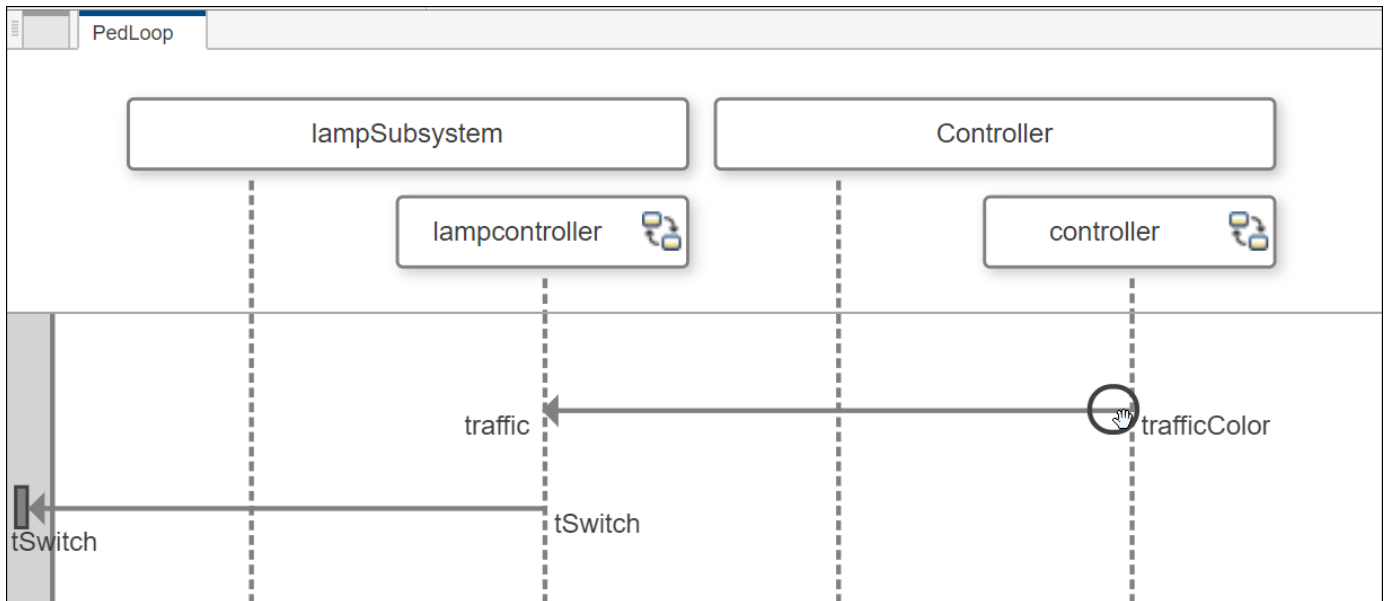
- 1 Draw a line from the controller lifeline to the lampcontroller lifeline. Start to type **traffic** in the **To** box, which fills in automatically as you type. Once the text fills in, select **traffic**.



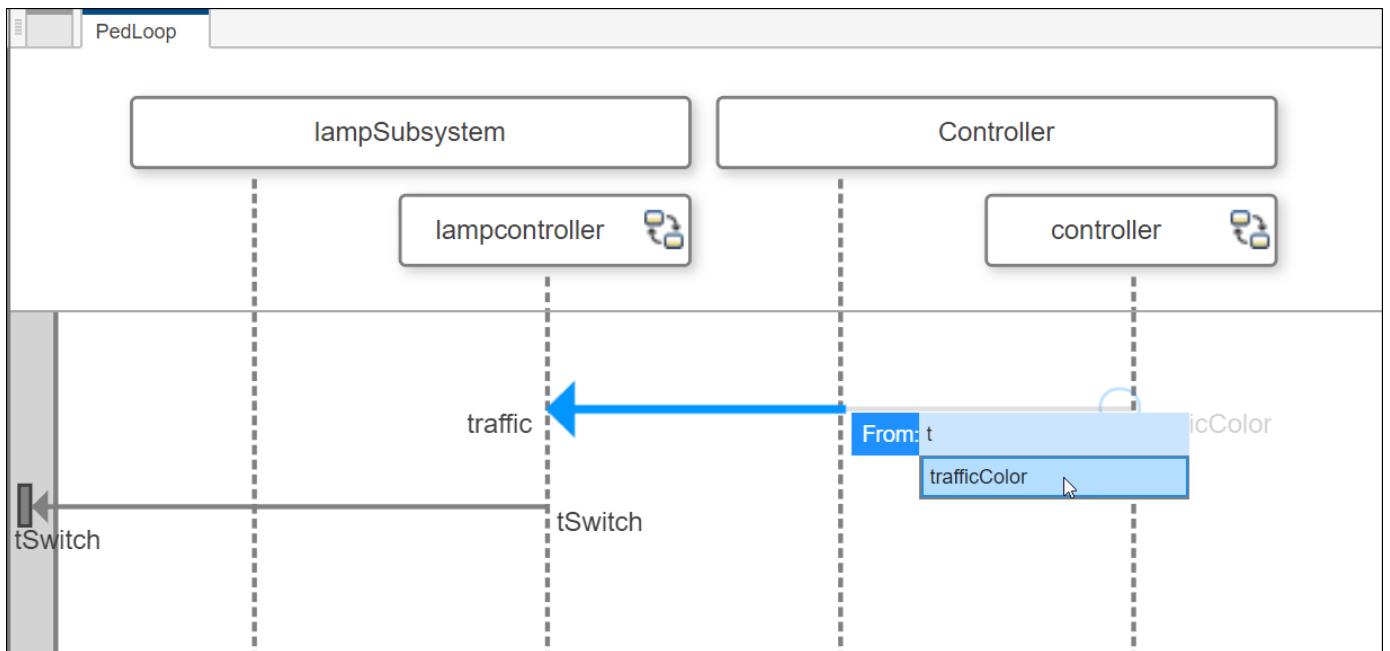
- 2 Since the trafficColor port and traffic port are connected in the model, a message is created from the traffic port to the trafficColor port in the sequence diagram.



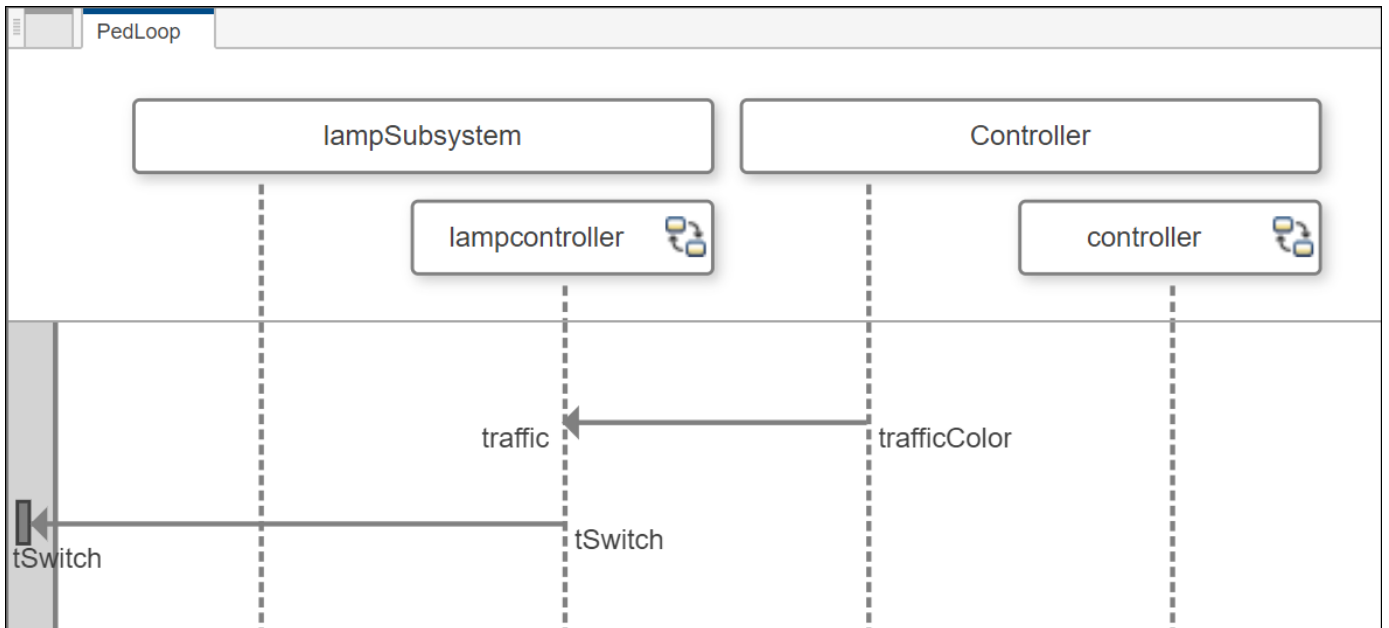
- 3 You can modify the source and destination of a message after the message has been created. Click the trafficColor message end to select it.



- 4 Click and drag the trafficColor message end to the Controller parent lifeline, then select the trafficColor port.



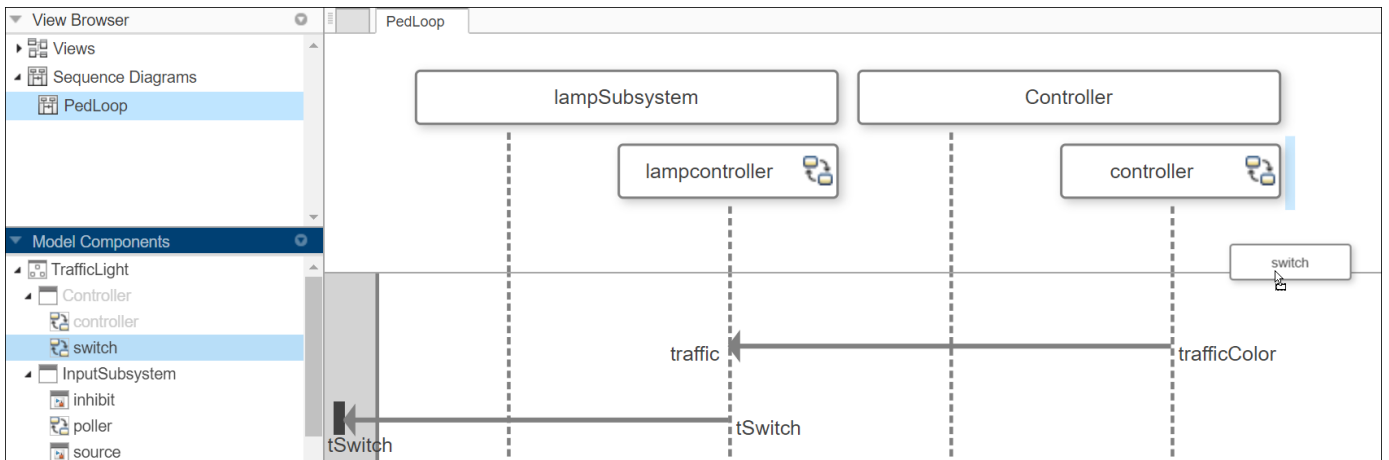
- 5 Once the trafficColor port is selected, the message end moves from the controller child lifeline to the Controller parent lifeline.



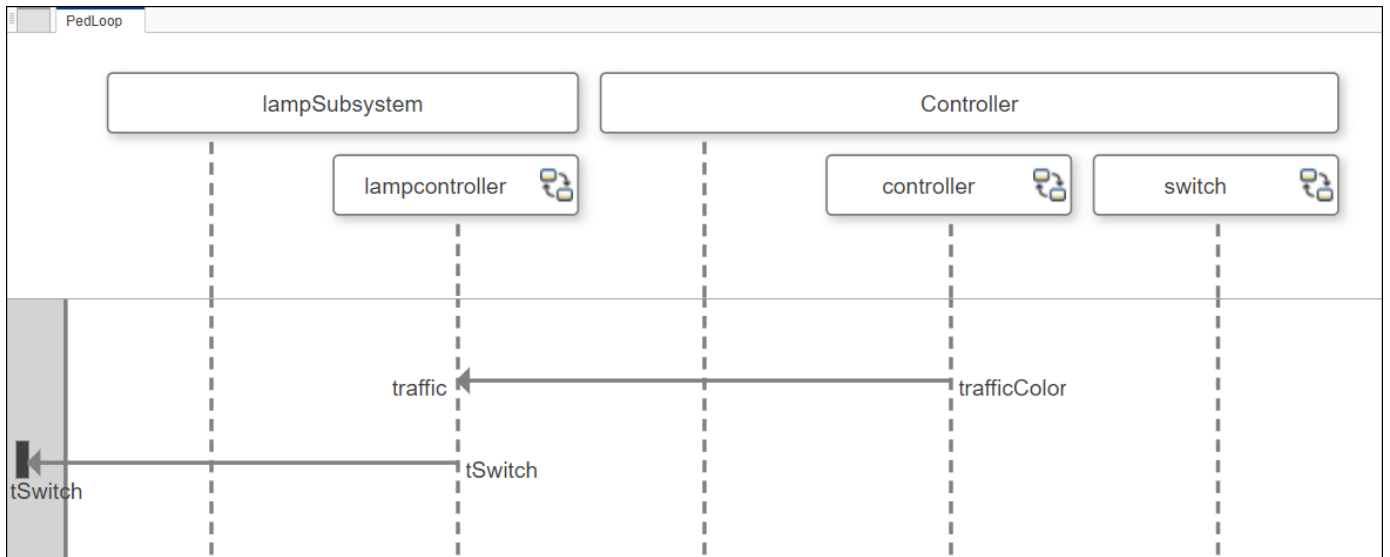
You can also rename message ends and the associated ports by double-clicking the name of a message end.

Modify Sequence Diagram Using Model Browser

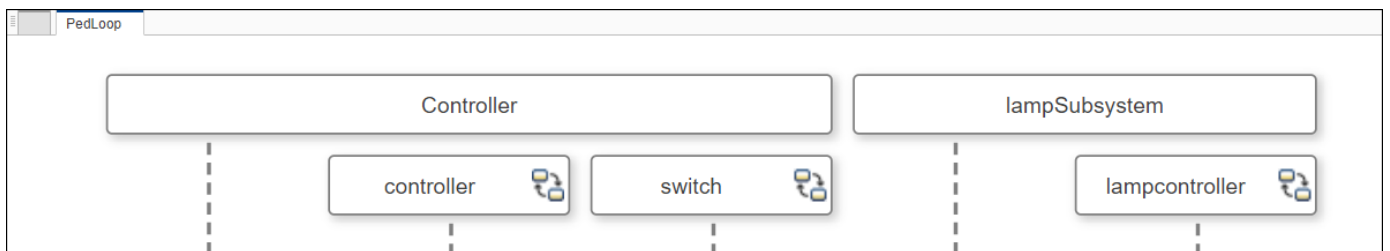
- 1 The Views Gallery model browser located on the bottom left of the canvas is called **Model Components**. Click and drag the switch child component into the sequence diagram.



- 2 The sequence diagram is updated with a new lifeline.

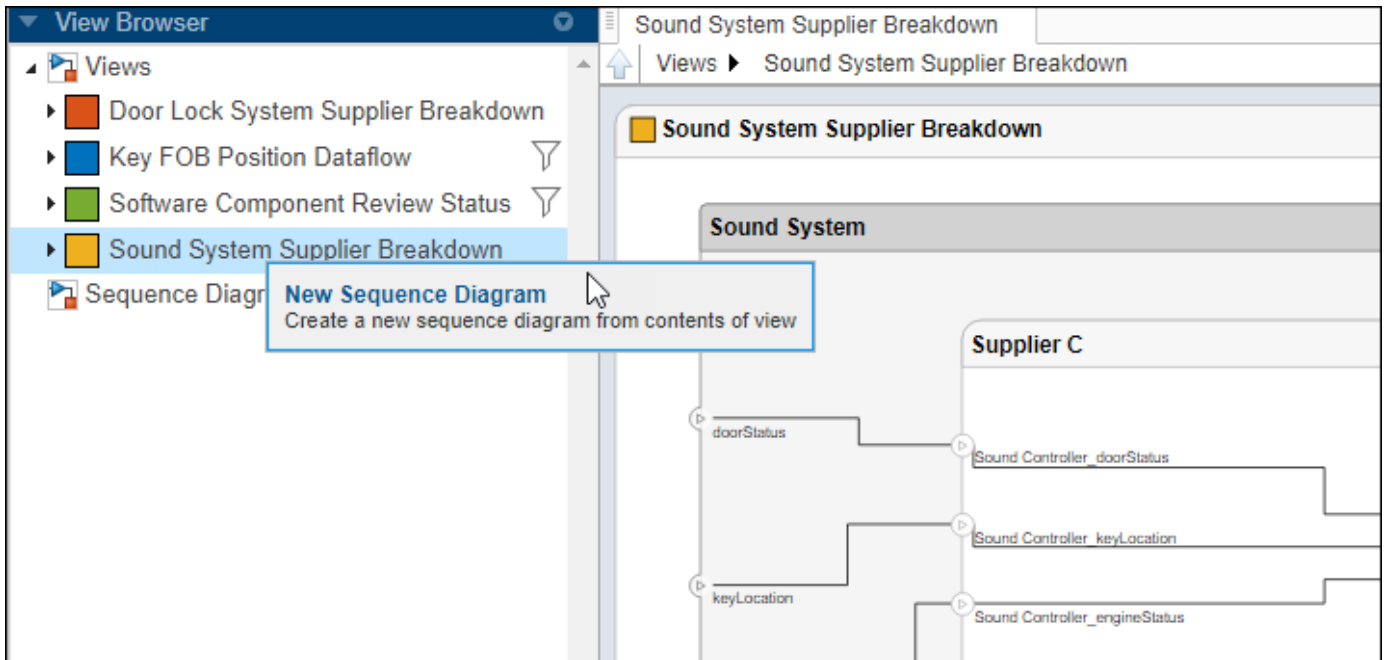


- 3 Click and drag to reorder the lampSubsystem and the Controller lifelines.

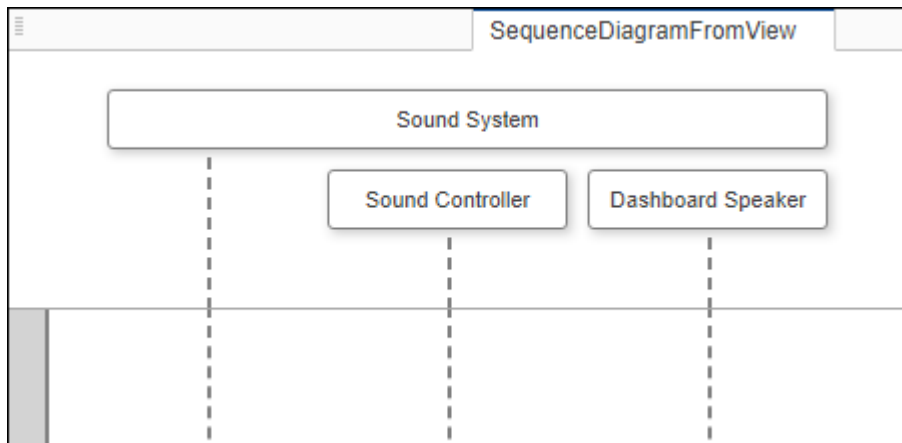


Create Sequence Diagram from View

- 1 In the MATLAB Command Window, enter `scKeylessEntrySystem`. The architecture model opens in the Simulink Editor.
- 2 To open the **Architecture Views Gallery** for the model, navigate to **Modeling > Views > Architecture Views**.
- 3 Right-click the Sound System Supplier Breakdown view and select **New Sequence Diagram**.



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



See Also

More About

- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Compose Architecture Visually” on page 1-2
- “Describe Component Behavior Using Simulink” on page 5-2
- “Describe Component Behavior Using Stateflow Charts” on page 5-23
- “Define Port Interfaces Between Components” on page 3-2

Implement Sequence Diagram Fragments

A sequence diagram describes an expected order of the events logged during execution of a Simulink model. There are four event types:

- 1 Signal read
- 2 Signal write
- 3 Message send
- 4 Message receive

These four events cover the two interaction styles described by sequence diagrams: signals and messages. You can specify events using messages between lifelines. Each message describes a pair of events. For signals, the pair of events is write followed by read. For messages, the pair of events is send followed by receive.

A fragment indicates how a group of messages within are executed or interact in a sequence diagram.

A fragment is used to model complex sequences, such as alternatives, in a message-based behavior diagram.

To add a fragment to a sequence diagram, click and drag around a group of messages or other fragments. A blue box appears. Pause on the ellipsis (...) menu to select from a list of possible fragments.



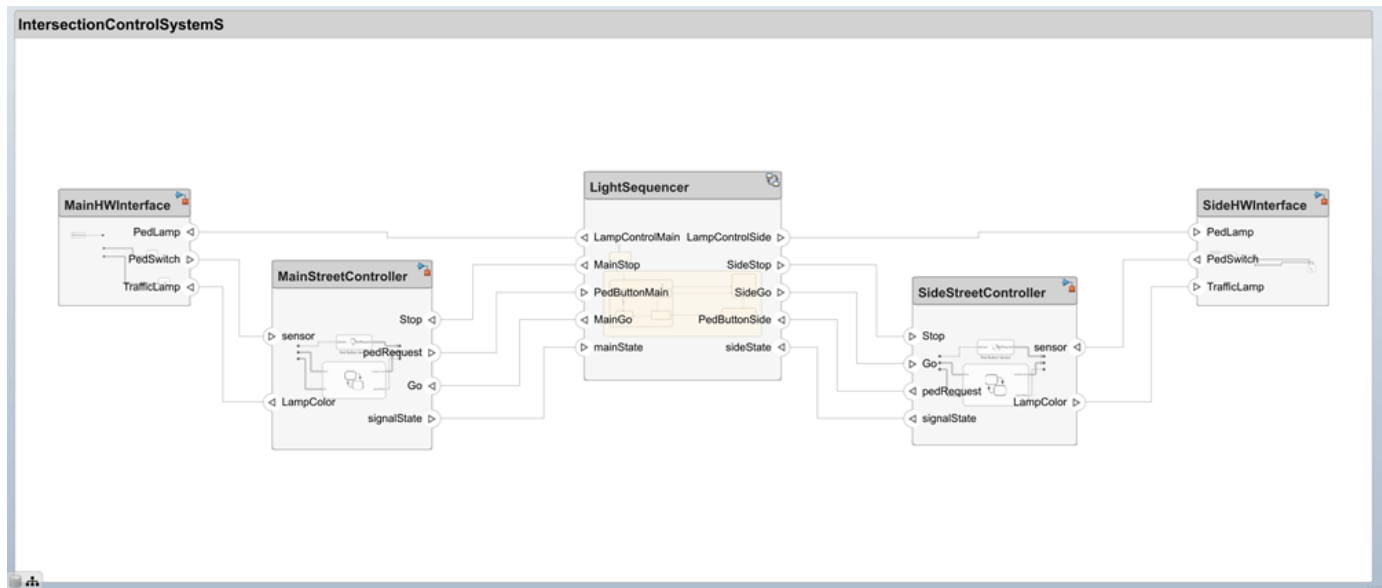
The fragment you select is added in the sequence diagram. For more information, see “Add Fragments and Operands” on page 5-45.

Sequence Diagram Fragments

This example shows an intersection control system to demonstrate how composite fragments are used in sequence diagrams.

Open the System Composer model that contains the sequence diagrams.

```
model = systemcomposer.openModel("IntersectionControlSystemS");
```



Open the Architecture Views Gallery to view the sequence diagrams.

```
openViews(model)
```

You can execute the model after setting these variables.

```
createWorkspaceVar("MainInputs",[0 3 22],[-1 1 -1]);
createWorkspaceVar("SideInputs",[0 50 52],[-1 1 -1]);
```

Fragment Semantics

Fragments in a sequence diagram execute messages in a specific order based on the kind of fragment. The following fragments are supported:

- Alternative (Alt) fragment
- Optional (Opt) fragment
- Loop (Loop) fragment
- Weak sequencing (Seq) fragment
- Strict sequencing (Strict) fragment
- Parallel (Par) fragment

An operand is a region in a fragment. Fragments have one or more operands depending on the kind of fragment. Operands can contain messages and additional fragments.

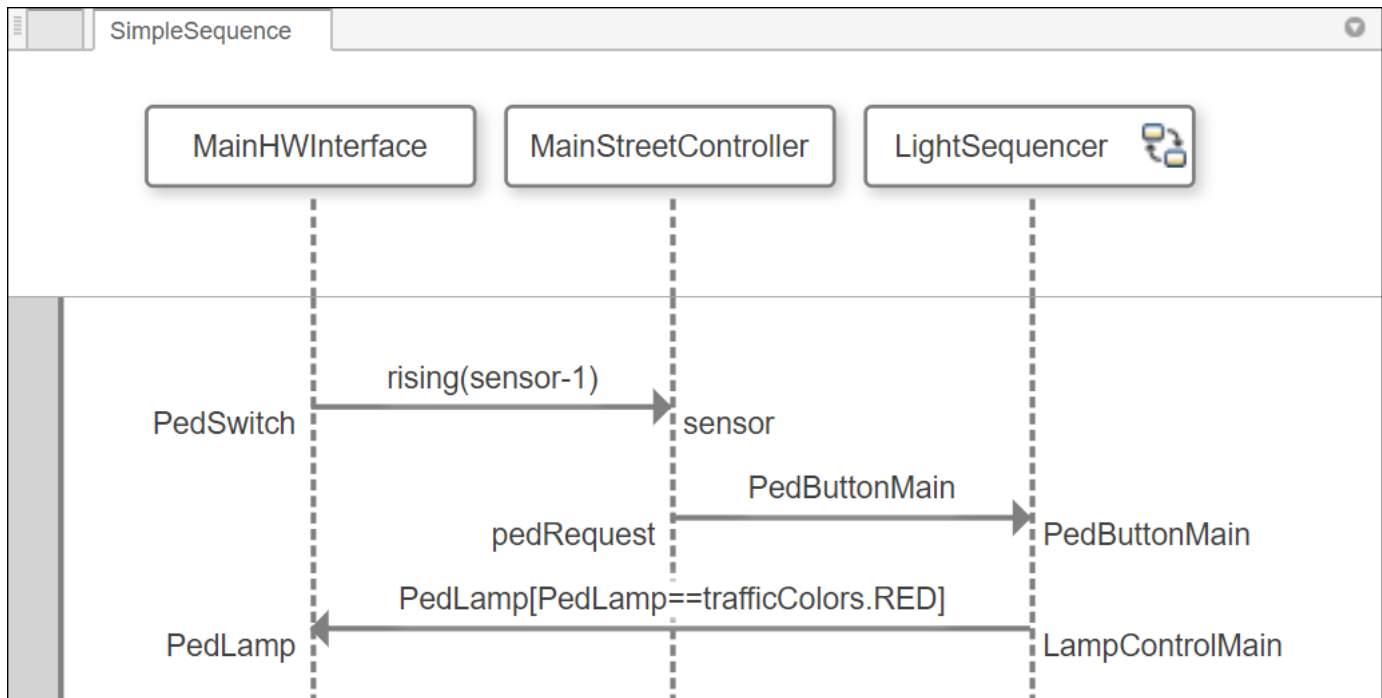
Each operand can include a constraint to specify whether the messages inside the operand execute. You can express the precondition of an operand as a MATLAB Boolean expression using the inputs of any lifeline.

Seq Fragment

The Seq fragment contains one operand and represents weak sequencing. The Seq fragment specifies the same ordering constraints as a sequence of messages with no fragment.

Weak sequencing means that the sequence of events on each lifeline is respected, but the sequencing of events between lifelines can differ from the order shown in the operand.

- On any given lifeline, any event must occur after the event immediately above it on the lifeline. If a message send or write event occurs from the same lifeline, the order is determined by how far down the lifeline the event occurs.
- The send or write event for a message must occur before the corresponding receive or read event.
- If two different message send and receive events or write and read events occur on the same lifeline, then the first message can be received before or after the second message is sent.

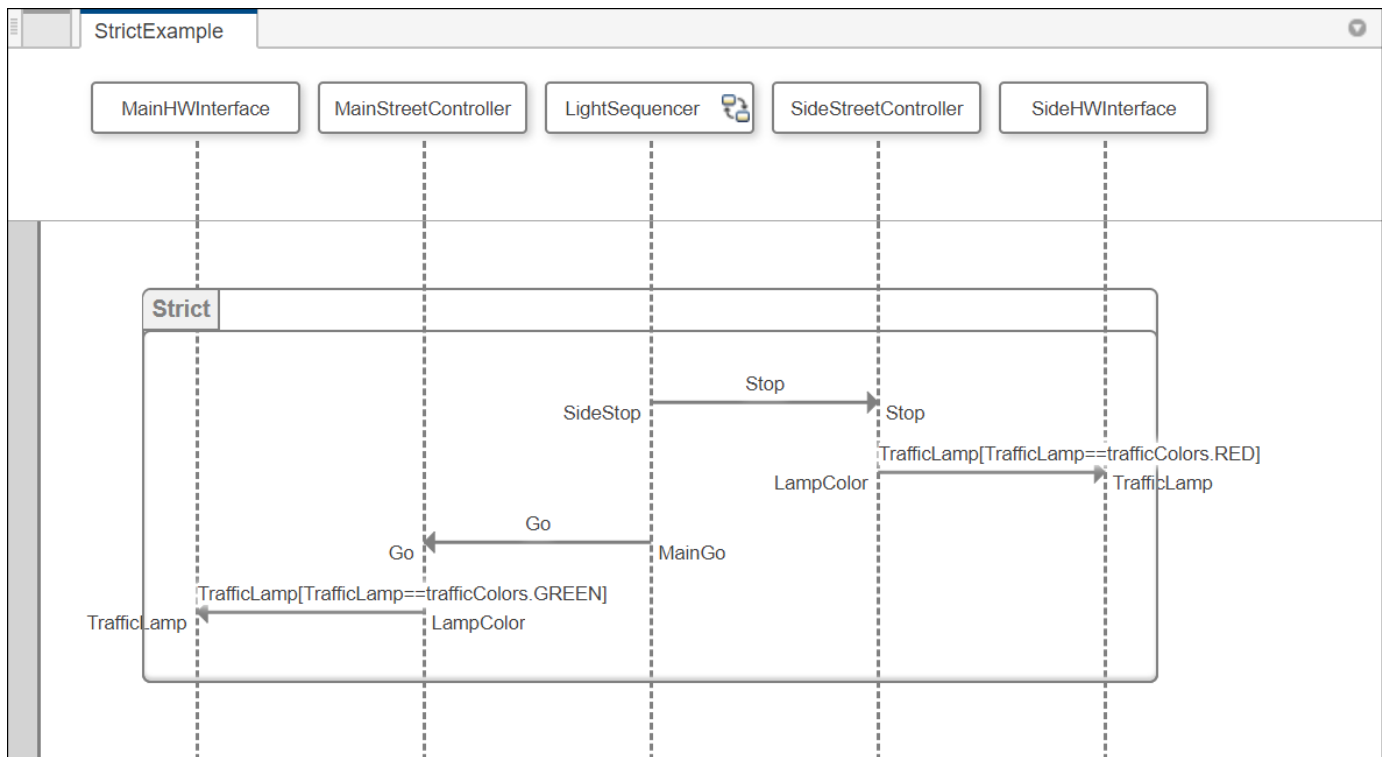


This sequence diagram specifies that after the sensor signal from its hardware interface rises through 1, the `MainStreetController` sends a `PedButtonMain` message to `LightSequencer`, which controls the intersection. `LightSequencer` then tells the hardware interface to turn the pedestrian lamp red, indicating that pedestrians should wait.

Strict Fragment

The `Strict` fragment contains one operand and represents strict sequencing.

Strict sequencing follows the weak sequencing found in a “Seq Fragment” on page 5-66 with the additional constraint that the order of the events in the operand be followed exactly. If messages are sent from the same lifeline, the order is determined by how far down the lifeline they are sent. Messages are received in the order on the operand regardless of which lifeline receives them.



This sequence diagram is an example of strict sequencing for an intersection controller. The traffic lights on the side street should be set to red before the traffic lights on the main street are set to green. Without strict sequencing, the order in which the street controllers perform their tasks would be undefined.

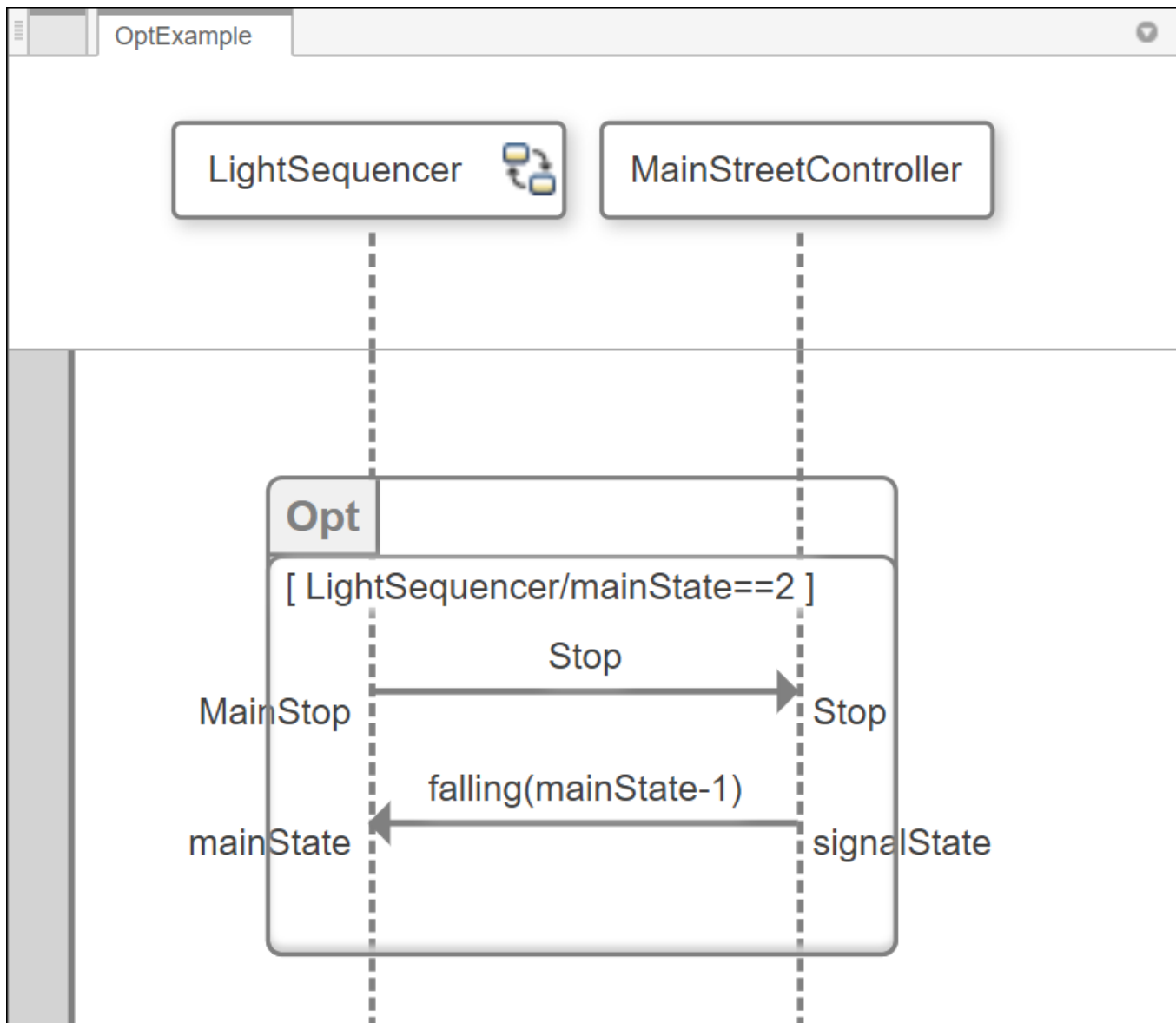
You can use the “Seq Fragment” on page 5-66 for a subset of the messages in a **Strict** fragment if all the messages do not require strict ordering.

Opt Fragment

The **Opt** fragment includes a single operand and indicates that the events described might occur or not. A constraint determines whether the group of messages are valid. If the group of messages are valid, they can execute, otherwise, this fragment is skipped. If no constraint is specified, the messages in the **Opt** fragment are executed only if the messages become valid.

The events in an **Opt** fragment occur if all of these conditions are met:

- The current event is a valid first event for the operand.
- The condition expressed by the operand constraint evaluates to **true**.



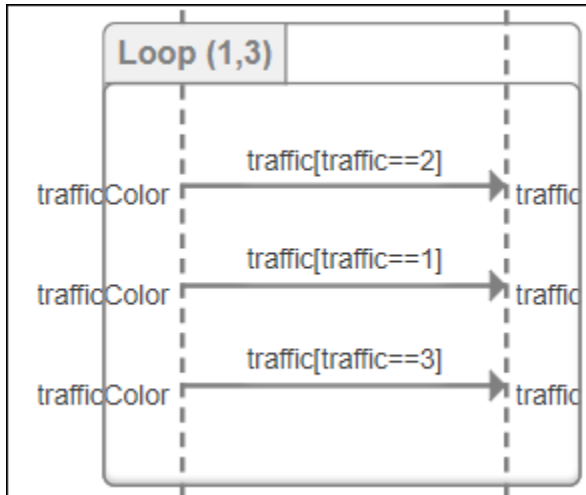
In this sequence diagram, the `LightSequencer` knows the state of each controller due to the state signal. The `LightSequencer` issues a `Stop` command to a controller if it is in the flowing (Green) state, which is indicated by the value 2. The operand in the `Opt` fragment is executed only if a `Stop` command is present and its `mainState` signal has the value 2. After the `Stop` command, the `mainState` signal falls to 1, indicating a stopping (Yellow) state.

Loop Fragment

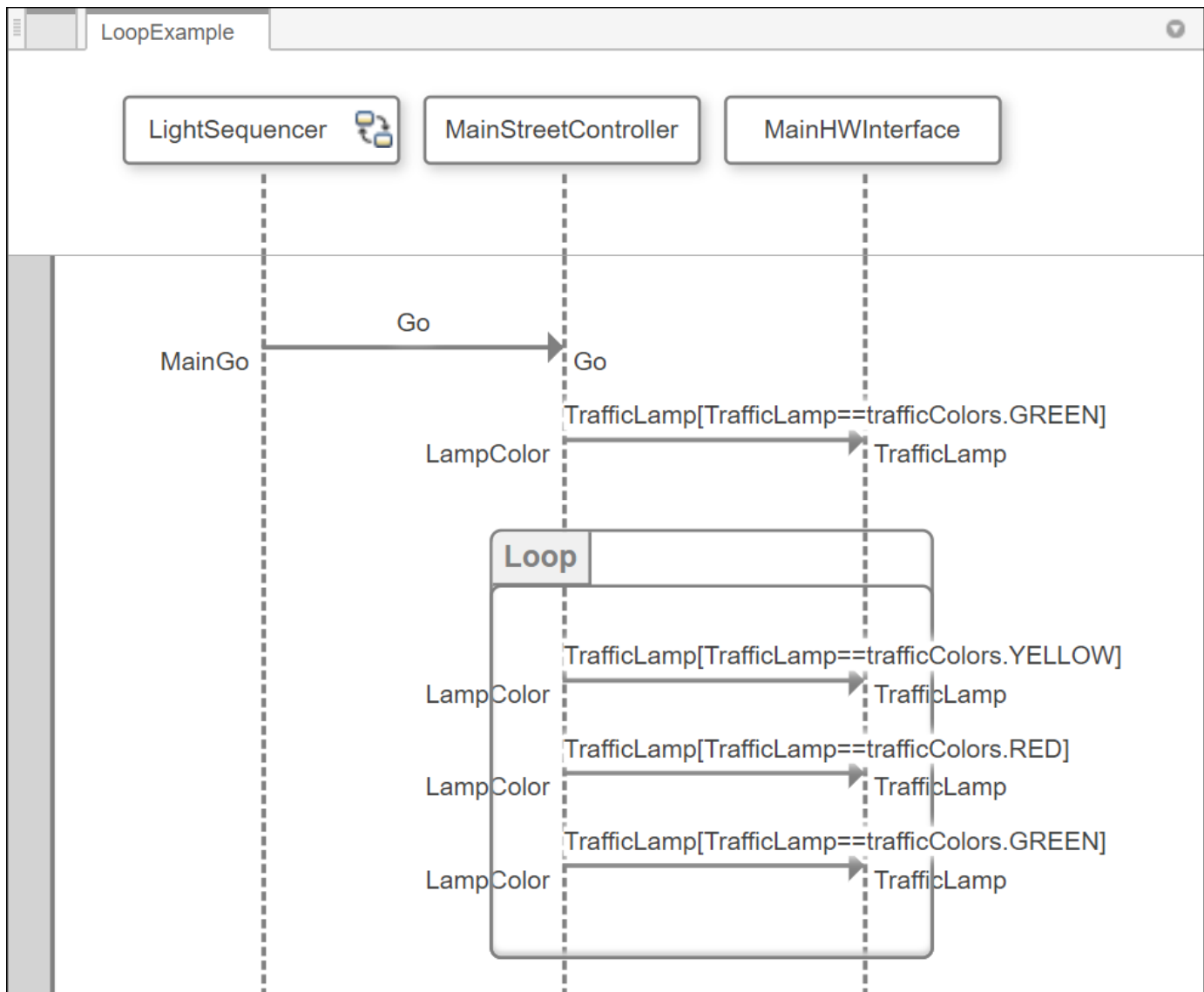
A `Loop` fragment has a single operand with a constraint and an expression describing the minimum (lower bound) and maximum (upper bound) number of times that the operand repeats. The upper bound can be `*`, indicating that there is no maximum. The lower bound can be `0`, indicating that the operand is optional. The default is a lower bound of `0` and an upper bound of `*`. With the default lower and upper bounds, the `Loop` fragment repeats a certain number of times according to the simulation time or the lower and upper bounds of the architecture model.

The single operand in the Loop fragment repeats if all of these conditions are met:

- The current event is a valid first event for the operand.
- The condition expressed by the operand constraint evaluates to `true`.
- The number of iterations is less than or equal to the upper bound.



The lower bound (1) is the minimum number of iterations of the loop. The upper bound (3) is the maximum. You can also define a constraint on a Loop fragment operand that determines whether the loop executes. When the Boolean expression is `false`, the loop is skipped.



This sequence diagram describes the default cycle of the main street traffic lamps. The `LightSequencer` issues a `Go` command so that the `MainStreetController` tells the `MainHWInterface` to turn the green lamps on. The controller then cycles the lights repeatedly through yellow, red, and green lamps, as indicated by the `Loop` fragment. The lower and upper bounds of the loop fragment are the default values of `0` and `*`, respectively, indicating that it allows any number of iterations.

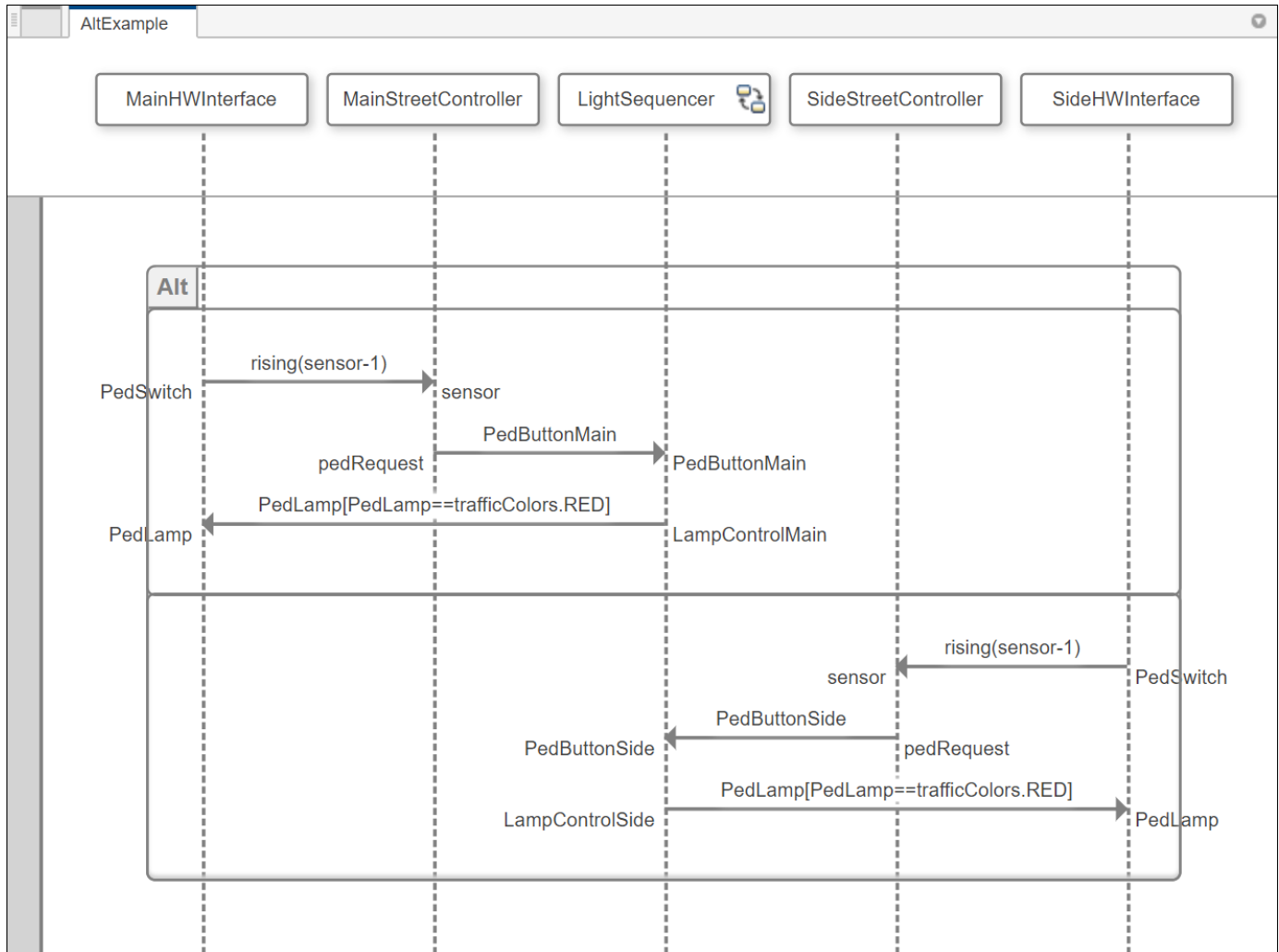
Alt Fragment

The `Alt` fragment is like an “`Opt` Fragment” on page 5-68 except that it has two or more operands, each describing a different message order.

The events for each operand in an `Alt` fragment occur if all of these conditions are met:

- The current event is a valid first event for the operand.

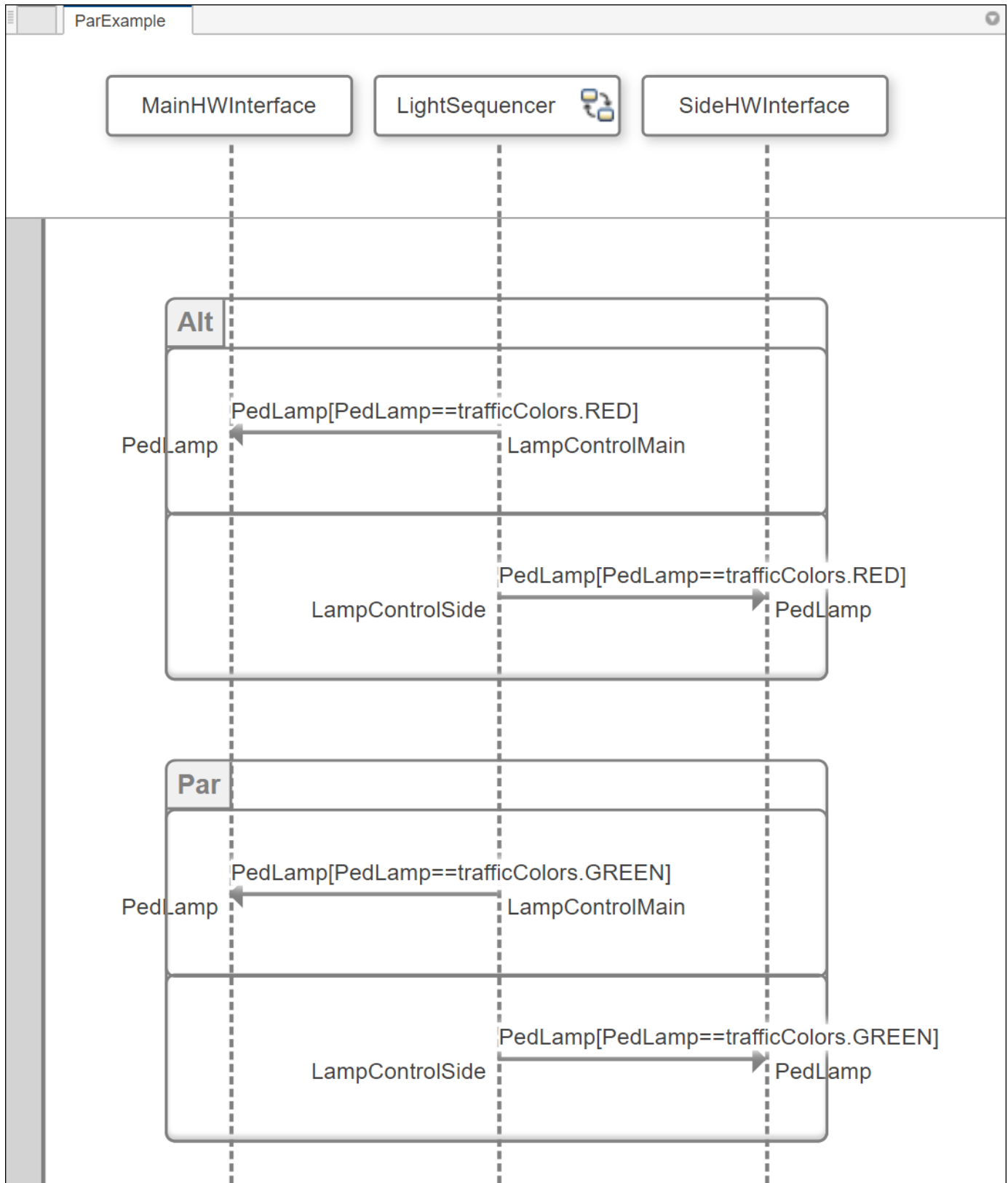
- The condition expressed by the operand constraint evaluates to `true`.



This sequence diagram shows that there are crossing buttons on the main street and the side street and either may be pressed. For a description of the first operand, see the sequence diagram for the “Seq Fragment” on page 5-66.

Par Fragment

Par stands for parallel. A **Par** fragment can have two or more operands. In a **Par** fragment, the order of events in each operand is enforced, but there is no constraint on the order of events in different operands. You should use **Par** fragments wherever order between messages is not important because this fragment imposes few constraints on system design.



No matter which crossing button the pedestrian presses, the controller stops traffic on both streets. This sequence diagram specifies this state using an **Alt** fragment for the red lamp color. The green light that indicates safety in crossing, is shown on both streets. The **Par** fragment indicates that both the main and side streets show the green color, but order does not matter.

See Also

More About

- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Describe Component Behavior Using Simulink” on page 5-2
- “Describe Component Behavior Using Stateflow Charts” on page 5-23
- “Describe Component Behavior Using Simscape” on page 5-75
- “Define Port Interfaces Between Components” on page 3-2

Describe Component Behavior Using Simscape

A physical subsystem is a Simulink subsystem with Simscape connections.

A physical subsystem with Simscape connections uses a physical network approach suited for simulating systems with real physical components and represents a mathematical model.

Using Simscape behaviors for components in System Composer improves model simulation and design for systems with physical components. This functionality requires a Simscape license. For more information, see “Basic Principles of Modeling Physical Networks” (Simscape).

You can simulate the Simscape component implementations in System Composer. Use the **Simulation Data Inspector** to view and compare simulation results between model designs.

To describe component behavior in Simscape for a System Composer architecture model, follow these steps:

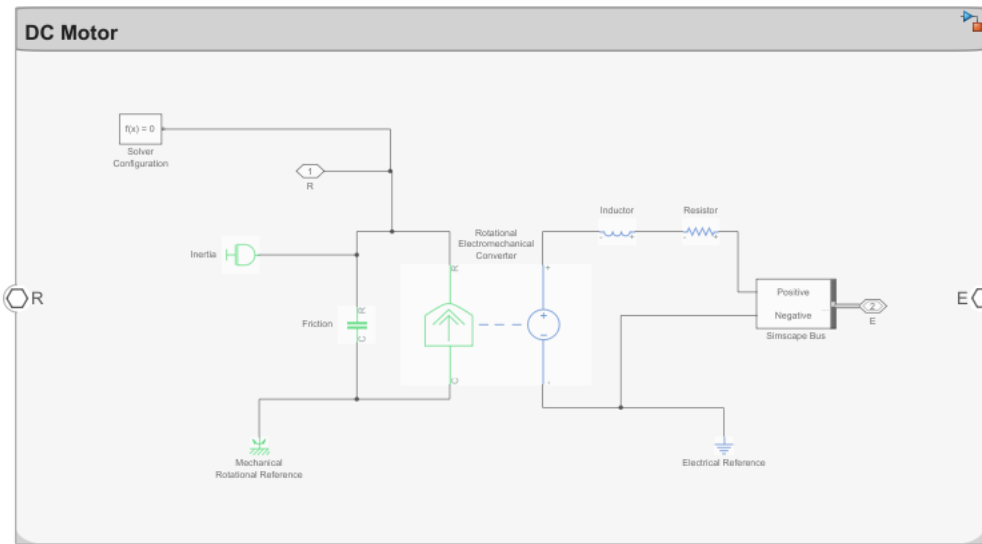
- 1 “Define Physical Ports on Component” on page 5-76
- 2 “Specify Physical Interfaces on Ports” on page 5-76
- 3 “Create Simulink Subsystem Component” on page 5-77
- 4 “Describe Component Behavior Using Simscape” on page 5-78

Open this model to interact with a System Composer architecture model named Fan with Simscape behavior on a component DC Motor. The steps in this tutorial will produce this model.

Note This example uses Simscape blocks. If you do not have a Simscape license, you can open and simulate the model but can only make basic changes, such as modifying block parameters.

Architecture Model with Simscape Behavior for a DC Motor

This example shows a DC motor in an architecture model of a fan. The DC motor is modeled using a Simscape behavior within a Simulink subsystem component.

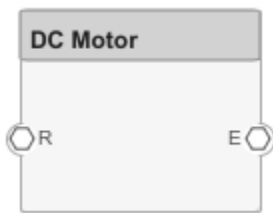


Define Physical Ports on Component

A physical port represents a Simscape physical modeling connector port called a Connection Port.

Use physical ports to connect components in an architecture model or to enable physical systems in a Simulink subsystem.

Create a new System Composer architecture model. Add a component called **DC Motor** to the canvas. To add physical ports to the component, pause on the boundary of the component until a port outline appears. Click the port outline and, from the options, select **Physical**.



Physical ports can also be used to connect to Simscape blocks.

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

Specify Physical Interfaces on Ports

You can specify physical interfaces on the physical ports.



A physical interface defines the kind of information that flows through a physical port. The same interface can be assigned to multiple ports. A physical interface is a composite interface equivalent to



a `Simulink.ConnectionBus` object that specifies any number of `Simulink.ConnectionElement` objects.

Use a physical interface to bundle physical elements to describe a physical model using at least one physical domain.

A physical element describes the decomposition of a physical interface. A physical element is equivalent to a `Simulink.ConnectionElement` object.

Define the **Type** of a physical element as a physical domain to enable use of that domain in a physical model.

- 1 To open the **Interface Editor**, navigate to **Modeling > Interface Editor**.
- 2 To add a new physical interface definition, click the list next to the  icon and select **Physical Interface**. Name the physical interface `ElectricalInterface`.
- 3 To add a physical element to the physical interface, click the  icon. Physical interface and physical element names must be valid MATLAB variable names. Create the physical elements `Positive` and `Negative`.
- 4 In the **Type** column, define the Simscape domain to which these physical elements belong. In this case, both belong to `foundation.electrical.electrical`.

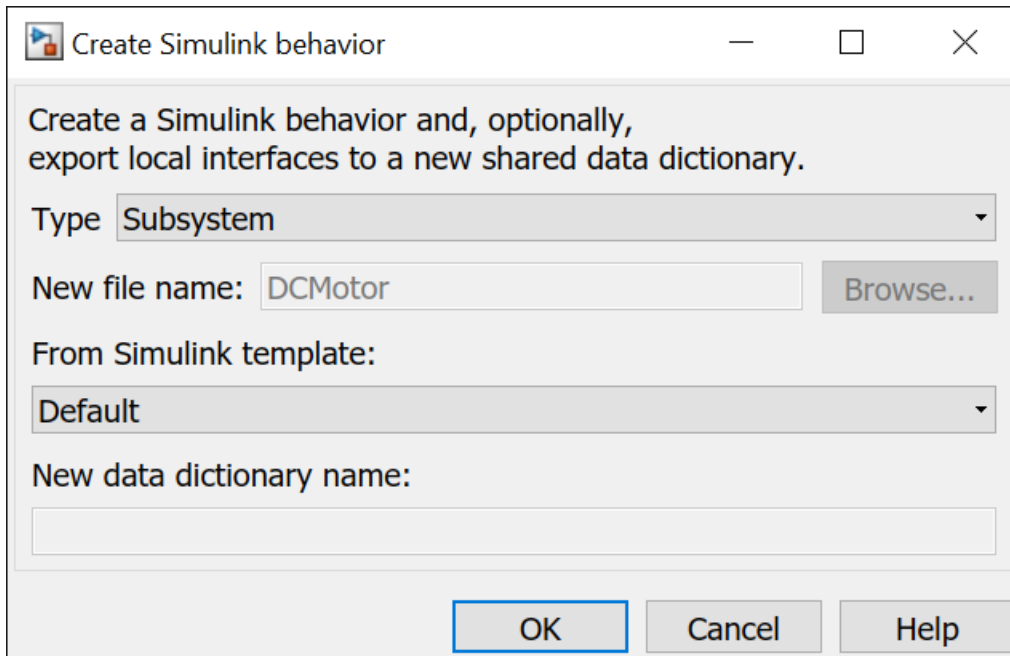
	Type
▼  Fan.slx	
▼  ElectricalInterface	
Positive	Connection: foundation.electrical.electrical
Negative	Connection: foundation.electrical.electrical

- 5 Select the E port on the DC Motor component. Right-click the `ElectricalInterface` physical interface on the **Interface Editor** and click **Assign to Selected Port(s)**.

Create Simulink Subsystem Component

You can create a Simulink subsystem in System Composer to enable direct Simscape integration. For more information, see “Create Simulink Subsystem Behavior Using Subsystem Component” on page 5-6.

Select the DC Motor component. Navigate to **Modeling > Create Simulink Behavior**, or use the right-click menu on the component.



Click **OK**.



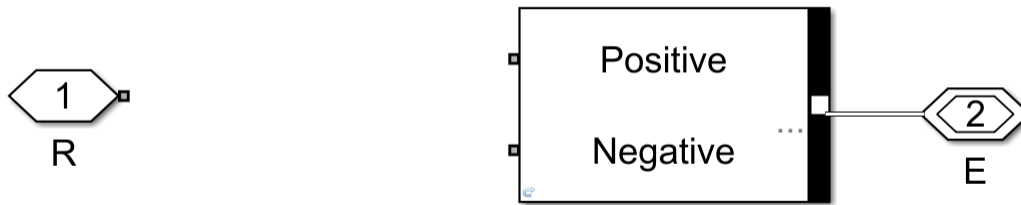
You can convert a subsystem component that is part of the parent System Composer model into a subsystem reference behavior then save and reuse the subsystem as a separate artifact. For more information, see “Convert Simulink Subsystem Component to Subsystem Reference Component” on page 5-8.

Describe Component Behavior Using Simscape

Double-click the subsystem component to describe component behavior using Simscape. For the DC motor this example is based on, see “Evaluating Performance of a DC Motor” (Simscape).

The physical interface can be decomposed into physical elements using a Simscape bus. Each physical element represents a conserving connection associated with a domain in Simscape. Simscape buses bundle conserving connections. For more information, see Simscape Bus (Simscape).

Add a Simscape Bus block next to the E physical port. Double-click the Simscape Bus and select the connection type Bus: ElectricalInterface. Connect the E physical port to the Simscape Bus block. The domain foundation.electrical.electrical defined under the **Type** of the Positive and Negative physical elements are used for any connections from these ports.



You can also use owned interfaces defined locally on ports to enable domain-specific lines on a Simscape behavior model in System Composer. Edit the port interface through the Property Inspector. Navigate to **Modeling > Property Inspector**. In this case, Simscape Bus blocks are not needed, and the port can connect directly to the physical connection of the specified domain. Add an owned physical interface to the physical port R with **Type** as a `foundation.mechanical.rotational.rotational` domain. Selecting **edit** to **Open in Interface Editor** enters the **Port Interface View** in the **Interface Editor**. For more information, see “Define Owned Interfaces Local to Ports” on page 3-11.

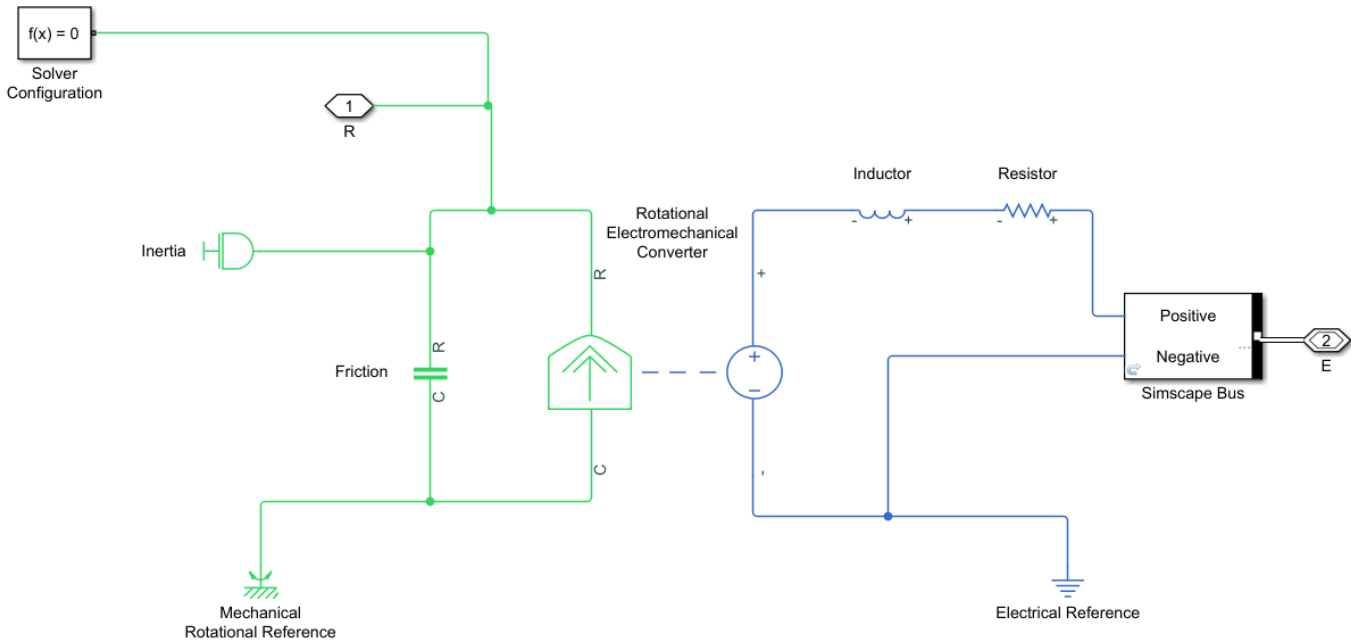
Property Inspector	
Port	
Architecture Info	
NAME	VALUE
▼ Main	
Name	R
▼ Interface	
Name	<owned>
Action	PHYSICAL
Open in Interface Editor	edit ...
Type	foundation.mechanical.rotational.rotational

Using the Library Browser, retrieve the following Simscape blocks and construct the DC Motor model with electrical and rotational mechanical domain-specific connectors.

A physical connector can represent a nondirectional conserving connection of a specific physical domain. Connectors can also represent physical signals.

Use physical connectors to connect physical components that represent features of a system to simulate mathematically.

For more information, see “Domain-Specific Line Styles” (Simscape).



Physical modeling uses the network approach and is therefore different from regular Simulink modeling. For more information, see “Modeling Best Practices” (Simscape) and “Troubleshooting Simulation Errors” (Simscape).

See Also

`createSimulinkBehavior` | `addPort` | `addPhysicalInterface` | `addElement` | `setInterface` | `createInterface`

More About

- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Describe Component Behavior Using Simulink” on page 5-2
- “Describe Component Behavior Using Stateflow Charts” on page 5-23
- “Define Port Interfaces Between Components” on page 3-2

Synchronize Sequence Diagrams and Architecture Models

This example shows how to maintain consistency between sequence diagrams and an architecture model.

Open the model.

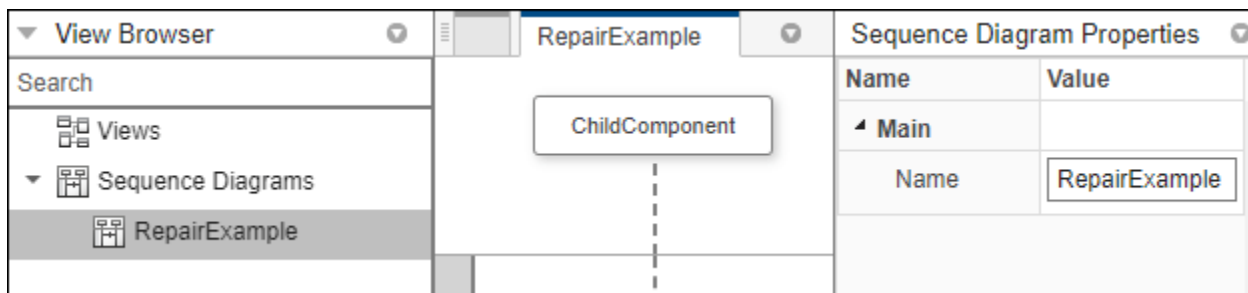
```
model = systemcomposer.openModel("mRetargetElements");
```

Open the Architecture Views Gallery.

```
openViews(model)
```

Pull Changes from Architecture Model to Sequence Diagram

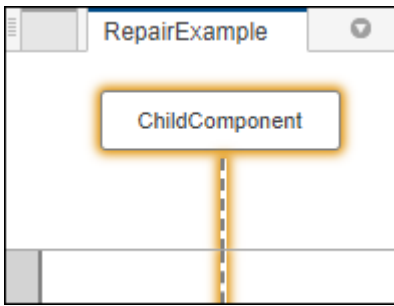
1. In the **View Browser**, select the RepairExample sequence diagram. Inspect the lifeline named ChildComponent.



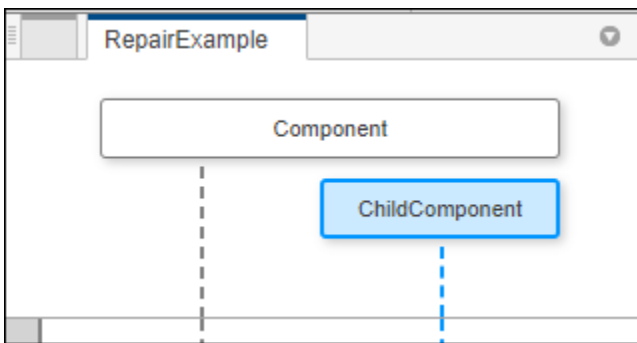
2. Return to the model canvas. Marquee select the ChildComponent component. Pause on the ellipses (...) menu and select Create Subsystem. Specify the name of the new component as Component. The ChildComponent component is now the child of the Component component.



3. Click **Check Consistency** on the sequence diagram RepairExample. The sequence diagram has become inconsistent and the ChildComponent lifeline is highlighted because it is no longer at the root level of the diagram.



4. Select the ChildComponent lifeline and, on the toolbar, click **Repair**. The sequence diagram RepairExample is updated after changes are pulled from the architecture model mRetargetElements.

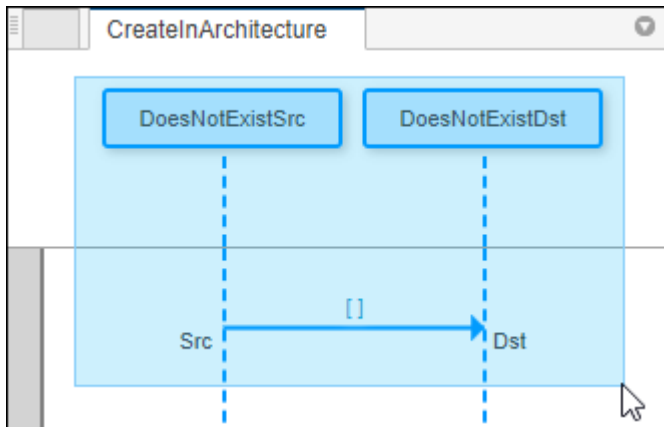


Push Changes from Sequence Diagram to Architecture Model

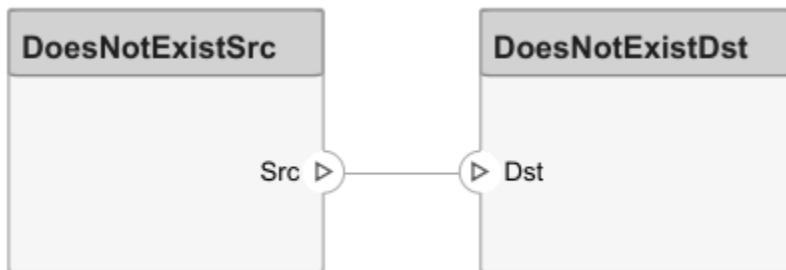
1. In the **View Browser**, select the CreateInArchitecture sequence diagram.

Sequence Diagram Properties	
Name	Value
Main	
Name	CreateInArchitecture

2. Marquee select the contents of the sequence diagram, including the two lifelines and the message.

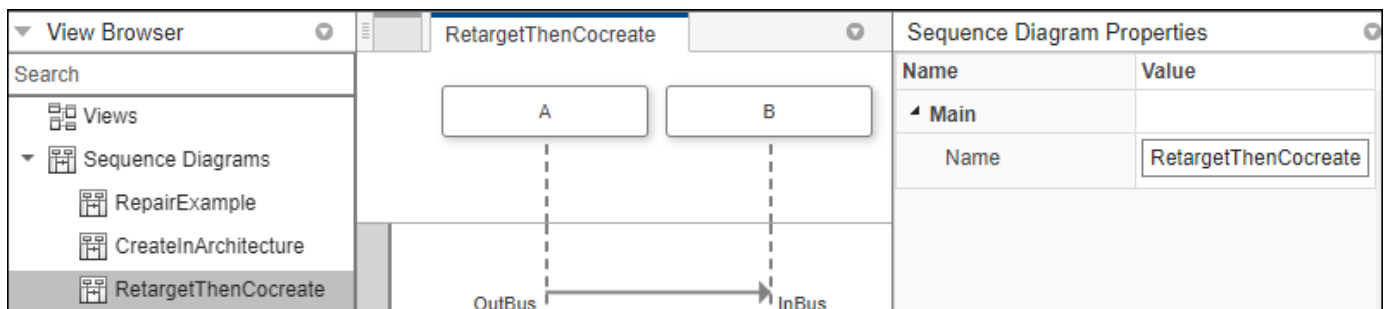


3. In the toolbar, click **Create in Architecture**. The architecture model `mRetargetElements` is updated after changes are pushed from the sequence diagram.

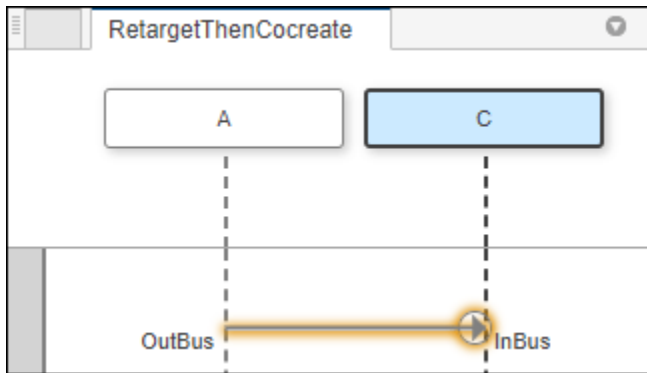


Retarget Lifeline and Create New Connection in Architecture Model

1. In the **View Browser**, select the `RetargetThenCocreate` sequence diagram.



2. Select the **B** lifeline on the sequence diagram and, from the **Architecture Element** menu, select **C** from the list. The sequence diagram becomes inconsistent, and the message is highlighted.



3. Select the message and, in the toolstrip, click **Create in Architecture**. A new connection is created in the architecture model `mRetargetElements`.



See Also

More About

- “Describe System Behavior Using Sequence Diagrams” on page 5-32
- “Compose Architecture Visually” on page 1-2
- “Author Sequence Diagram for Traffic Light Example” on page 5-38
- “Implement Sequence Diagram Fragments” on page 5-65
- “Use Sequence Diagrams with Architecture Models” on page 5-51

Allocate Architecture Models

- “Create and Manage Allocations” on page 6-2
- “Allocate Architectures in Tire Pressure Monitoring System” on page 6-5
- “Systems Engineering Approach for SoC Applications” on page 6-10

Create and Manage Allocations

This example shows how to create and manage System Composer™ allocations.

An *allocation* establishes a directed relationship from architectural elements — components, ports, and connectors — in one model to architectural elements in another model. Resource-based allocation allows you to allocate functional architectural elements to logical architectural elements and logical architectural elements to physical architectural elements. Another common use case for allocations is to establish relationships from software components to hardware components to indicate a deployment strategy.

An *allocation scenario* contains a set of allocations between a source and a target model. Allocate between model elements within an allocation in an allocation scenario. The default allocation scenario is called `Scenario 1`.

An *allocation set* consists of one or more allocation scenarios that describe various allocations between a source and a target model. Create an allocation set with allocation scenarios in the Allocation Editor.

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.

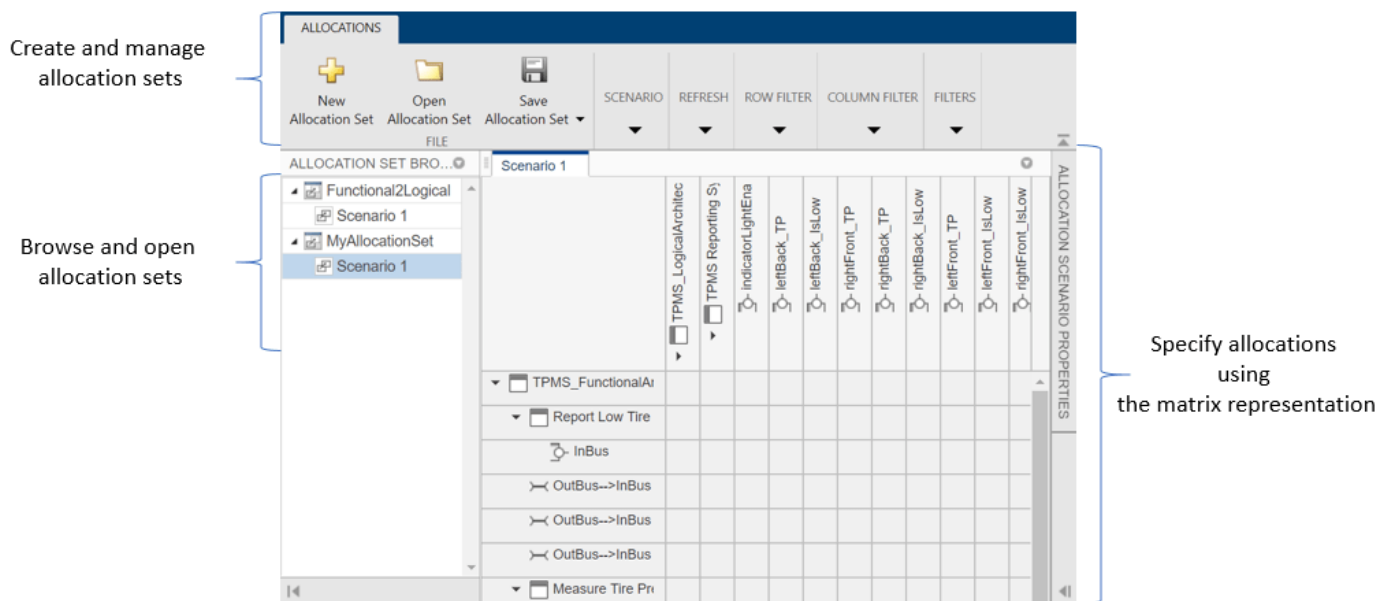
```
allocSet = systemcomposer.allocation.createAllocationSet(...  
    'Functional2Logical', ...% Name of the allocation set  
    'TPMS_FunctionalArchitecture', ... % Source model  
    'TPMS_LogicalArchitecture' ... % Target model  
    );
```

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 Lev
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);
```

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

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

See Also

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

More About

- “Manage Requirements” on page 2-8
- “Analyze Architecture” on page 7-2
- “Allocate Architectures in Tire Pressure Monitoring System” on page 6-5
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

Allocate Architectures in Tire Pressure Monitoring System

Use allocations to analyze a tire pressure monitoring system.

Overview

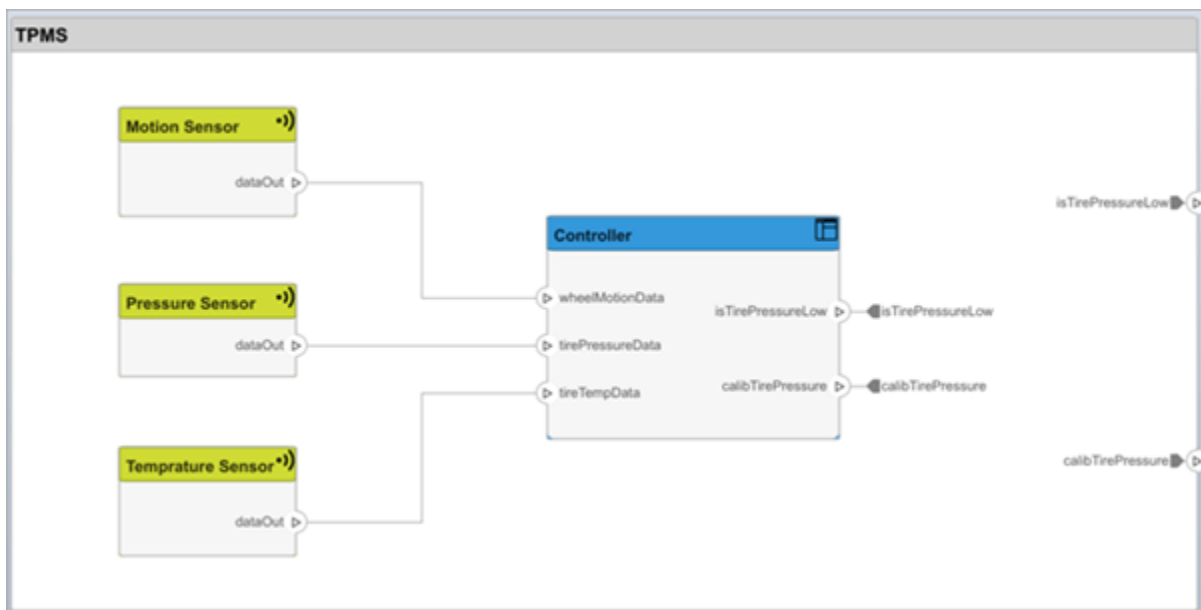
In systems engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system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

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

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

Use this command to open the project.

```
scExampleTirePressureMonitorSystem
```



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

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

Analyze Suppliers Providing Functions

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

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

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

```
allocTable
```

allocTable=8x4 table

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

Analyze Software Deployment Strategies

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

Get the platform architecture.

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

Load the allocation.

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

```

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

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

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

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

```

Build a table to showcase the results.

```

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

```

softwareDeploymentTable=6x2 table

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

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

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

```

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

```

end

See Also

getAllocatedTo | load | getScenario | getAllocatedFrom | synchronizeChanges |
getEvaluatedPropertyValue | systemcomposer.loadModel | find | getQualifiedName |
lookup

More About

- “Create and Manage Allocations” on page 6-2
- “Analyze Architecture” on page 7-2
- “Analysis Function Constructs” on page 7-9
- “Organize System Composer Files in Projects” on page 10-2
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

Systems Engineering Approach for SoC Applications

This example shows how to design a sample signal detector application on a System on Chip (SoC) platform using a systems engineering approach. The workflow in this example maps the application functions onto the selected hardware architecture.

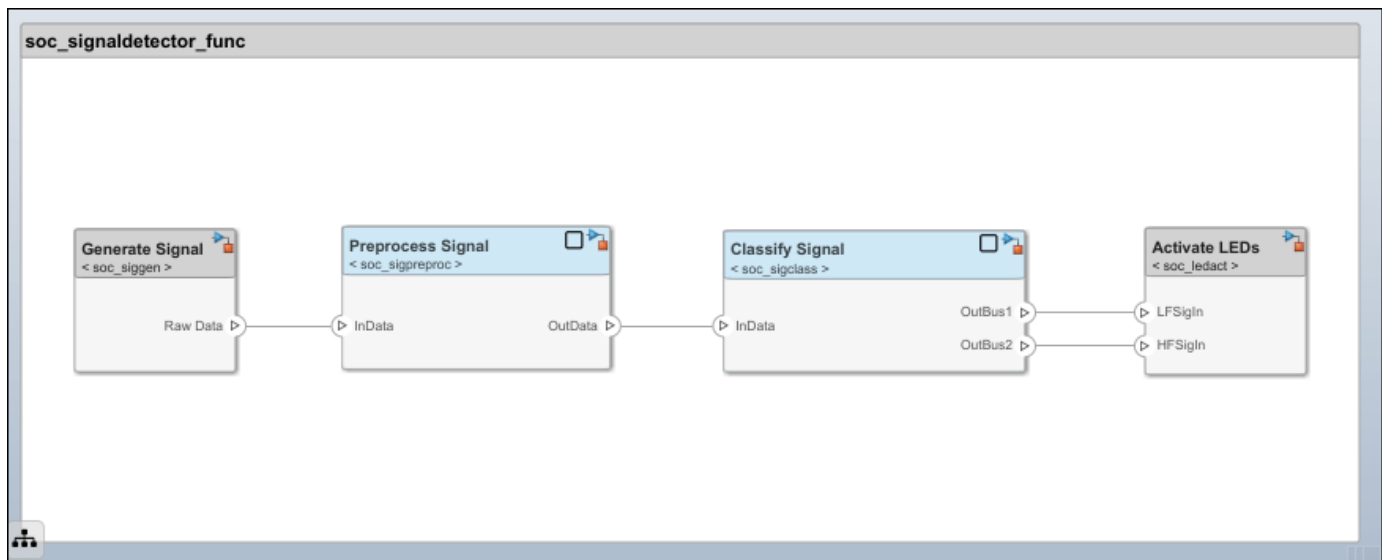
The signal detector application continuously processes the signal data and classifies the signal as either high or low frequency. The signal cannot change between high- and low-frequency classes faster than 1 ms. The signal is sampled at the rate of 10 MHz.

Functional Architecture

Define the functional architecture of the application. At this stage, the implementation of the application components is not known. You can use the System Composer™ software to capture the functional architecture.

This model represents the functional architecture with its main software components and their connections.

```
systemcomposer.openModel('soc_signaldetector_func');
```



The functional architecture of the application consists of these top-level components:

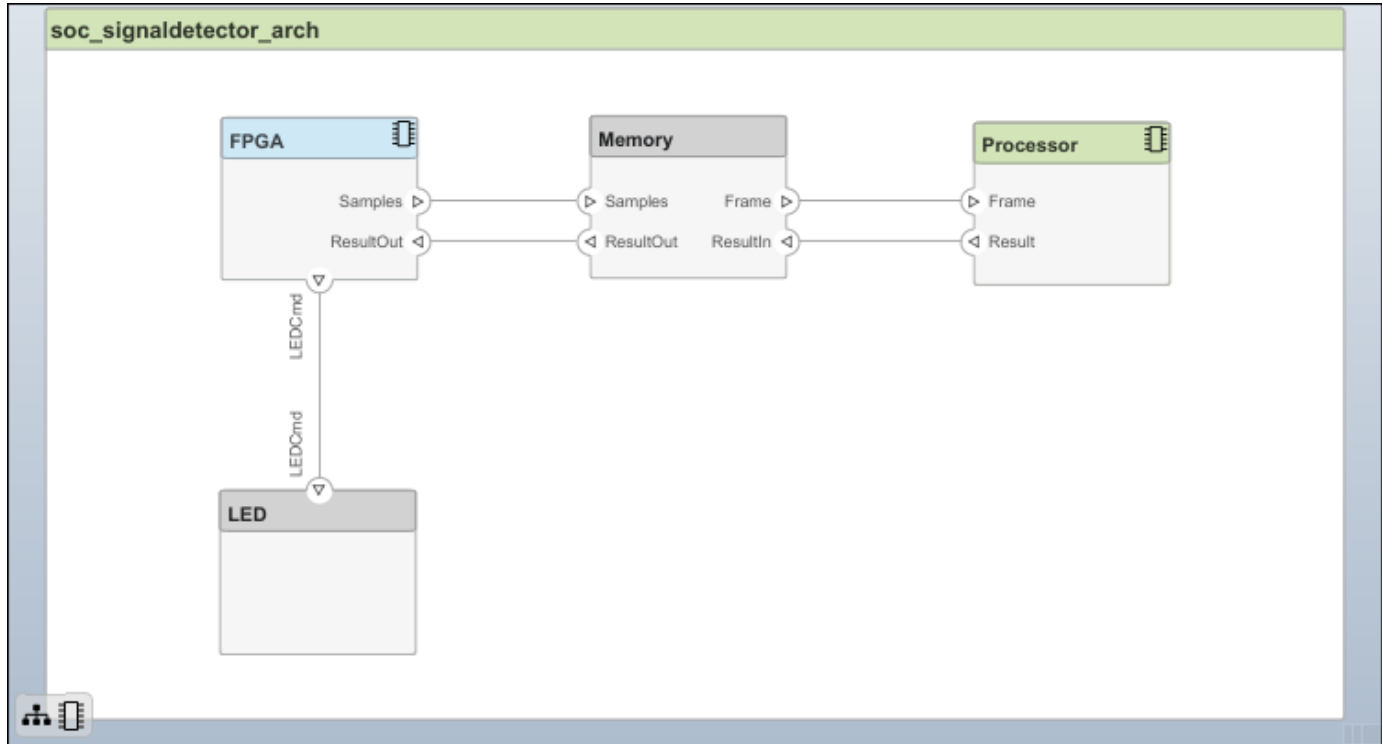
- 1 Generate Signal
- 2 Preprocess Signal
- 3 Classify Signal
- 4 Activate LEDs

Hardware Architecture

Select the hardware architecture. Due to the anticipated application complexity, choose an SoC device. The chosen SoC device has a hardware programmable logic (FPGA) core and an embedded processor (ARM) core. You can use the System Composer software to capture the details of the hardware architecture.

This model represents the hardware architecture with its main hardware components and their connections.

```
systemcomposer.openModel('soc_signaldetector_arch');
```



Behavioral Modeling

If the implementations for functional components are available, you can add them to the functional architecture as behaviors. In System Composer, for each functional component, you can link the implementation behaviors as Simulink® models. To review the component implementations, double-click each component in the functional architecture model.

After you define the behavior of each component, you can simulate the behavior of the entire application and verify its functional correctness. Select **Run** in the functional architecture model. Then, analyze the signals classification results in the **Simulation Data Inspector**. To change the signal type, select the **Generate Signal** component and then select the **Manual Switch** block. Confirm that the source signal is classified correctly.

Allocation of Functional and Hardware Elements

After refining the functional and hardware architecture, allocate different functional components to different hardware elements to meet desired system performance benchmarks. In this case, some functional components are constrained as to where in the hardware architecture they can be implemented. You must implement the **Generate Signal** and **Activate LEDs** components on the FPGA core in the chosen hardware architecture due to input output (I/O) connections. Comparatively, you can implement the **Preprocess Signal** and **Classify Signal** components on either the FPGA or on the processor core.

Component	Constraint
Generate Signal	FPGA

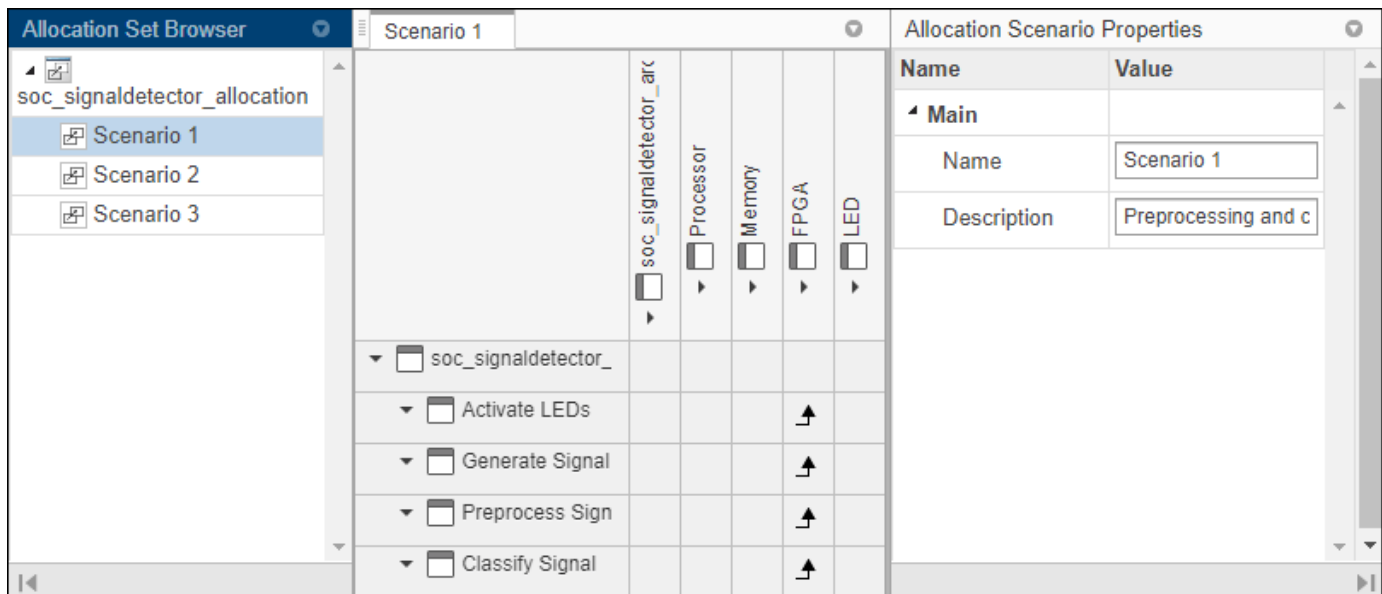
Preprocess Signal	-
Classify Signal	-
Activate LEDs	FPGA

This example shows how to use three possible scenarios for allocating the application functional architecture to the hardware architecture.

- The FPGA handles preprocessing and classification.
- The FPGA handles preprocessing and the processor handles classification.
- The processor handles preprocessing and classification.

System Composer captures these scenarios as Scenario 1, Scenario 2, and Scenario 3 using the Allocation Editor.

```
allocSet = systemcomposer.allocation.load('soc_signaldetector_allocation');
systemcomposer.allocation.editor
```



Choosing an allocation scenario requires finding an implementation that optimally meets the application requirements. Often you can find this implementation via static analysis without detailed simulation. In this example, use static analysis to analyze the computational costs of implementing different functional components on the processor and on the FPGA.

Implementation Cost

The implementation cost of a component depends on the required computation operations. To determine the implementation costs, consider these typical approaches.

- Component implementation is not available: Obtain the computational cost from the available reference implementations.
- The implementation and the hardware are available: Measure or profile the implementation cost on the candidate hardware.
- The implementation is available, but the hardware is not: Estimate the implementation cost by using the SoC Blockset™ algorithm analyzer function `socAlgorithmAnalyzerReport`.

The `socModelAnalyzer` function estimates the number of operations in a Simulink model and generates an algorithm analyzer report. To get the number of operations that a model executes to then analyze the implementation cost on the processor, use the dynamic analysis function option. To get the number of operators an algorithm requires to then analyze the implementation cost on the FPGA, use the static analysis function option. For an example on how to use `socModelAnalyzer`, see this sample function.

```
soc_signaldetector_costanalysis
```

```
*** Component: 'Preprocess Signal'
                                ADD(+)   MUL(*)
                                _____
FPGA Implementation             15        16
Processor Implementation        15300    16320
```

```
*** Component: 'Classify Signal'
                                ADD(+)   MUL(*)
                                _____
FPGA Implementation             32        18
Processor Implementation        32640    18360
```

The implementation costs for each functional component obtained in this code are entered in the corresponding stereotypes in the functional architecture. To verify the values, select each component in the functional architecture model and use the Property Inspector.

To learn more about `socModelAnalyzer`, see the “Compare FIR Filter Implementations Using `socModelAnalyzer`” (SoC Blockset) example. This example shows how to analyze the computational complexity of different implementations of a Simulink algorithm.

Allocation Choice

You can use the number of operators or operations that are required for implementing the application functional components to decide how to allocate the functional components to the hardware components. Analyze the candidate allocations by comparing the implementation cost against the available resources of the FPGA and the processor. This example uses sample values in the FPGA and the processor components in the hardware architecture model for the available computation resources. Verify the values by using the Property Inspector.

Typically, the analysis does not use the number of operators or operations directly. Rather, the number of operators or operations are multiplied by the cost of each operator or operation first. The cost of the operator or operations is hardware dependent. Determining such costs is beyond the scope of this example.

For an example on how to use the cost models, use this function. Observe that we require the capacity of the FPGA and the processor be greater than the estimated implementation cost as well as that the processor headroom be between 60 and 90 %.

```
soc_signaldetector_partitionanalysis
```

```
                                FPGA DSPs Used (out of 900)   FPGA LUT Used (out of 218600)   Processor Inst
```

Scenario 1	34	576
Scenario 2	16	192
Scenario 3	0	0

Based on the results Scenario 2 is feasible.

Data Path Design Between FPGA and Processor

The FPGA processes data sample-by-sample, and the processor processes frame-by-frame. Because the duration of a processor task can vary, to prevent data loss, a queue is needed to hold the data between the FPGA and processor. In this case you must set these parameters that are related to the queue: frame size, number of frame buffers, and FIFO size (that is, the number of samples in the FIFO). Also, in embedded applications, the task durations can vary between different task instances (for example, due to different code execution paths or due to variations in OS switching time). As a result, data might be dropped in the memory channel. The “Streaming Data from Hardware to Software” (SoC Blockset) example shows a systematic approach to choosing the previously mentioned parameters that satisfy the application requirements.

See Also

`socAlgorithmAnalyzerReport` | `socModelAnalyzer` | `systemcomposer.allocation.editor`

More About

- “Using the Algorithm Analyzer Report” (SoC Blockset)
- “Create and Manage Allocations” on page 6-2
- “Analyze Architecture” on page 7-2
- “Compose Architecture Visually” on page 1-2
- “Describe Component Behavior Using Simulink” on page 5-2

Analyze Architecture Model

- “Analyze Architecture” on page 7-2
- “Analysis Function Constructs” on page 7-9
- “Battery Sizing and Automotive Electrical System Analysis” on page 7-14
- “Calculate Endurance Using Quadcopter Architectural Design” on page 7-16
- “Design Insulin Infusion Pump Using Model-Based Systems Engineering” on page 7-23

Analyze Architecture

Perform static analysis on a System Composer architecture to evaluate characteristics of the system.

Analysis is a method for quantitatively evaluating an architecture for certain characteristics. Static analysis analyzes the structure of the system. Static analysis uses an analysis function and parametric values of properties captured in the system model.

Use analyses to calculate overall reliability, mass roll-up, performance, or thermal characteristics of a system, or to perform a SWaP analysis.

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:

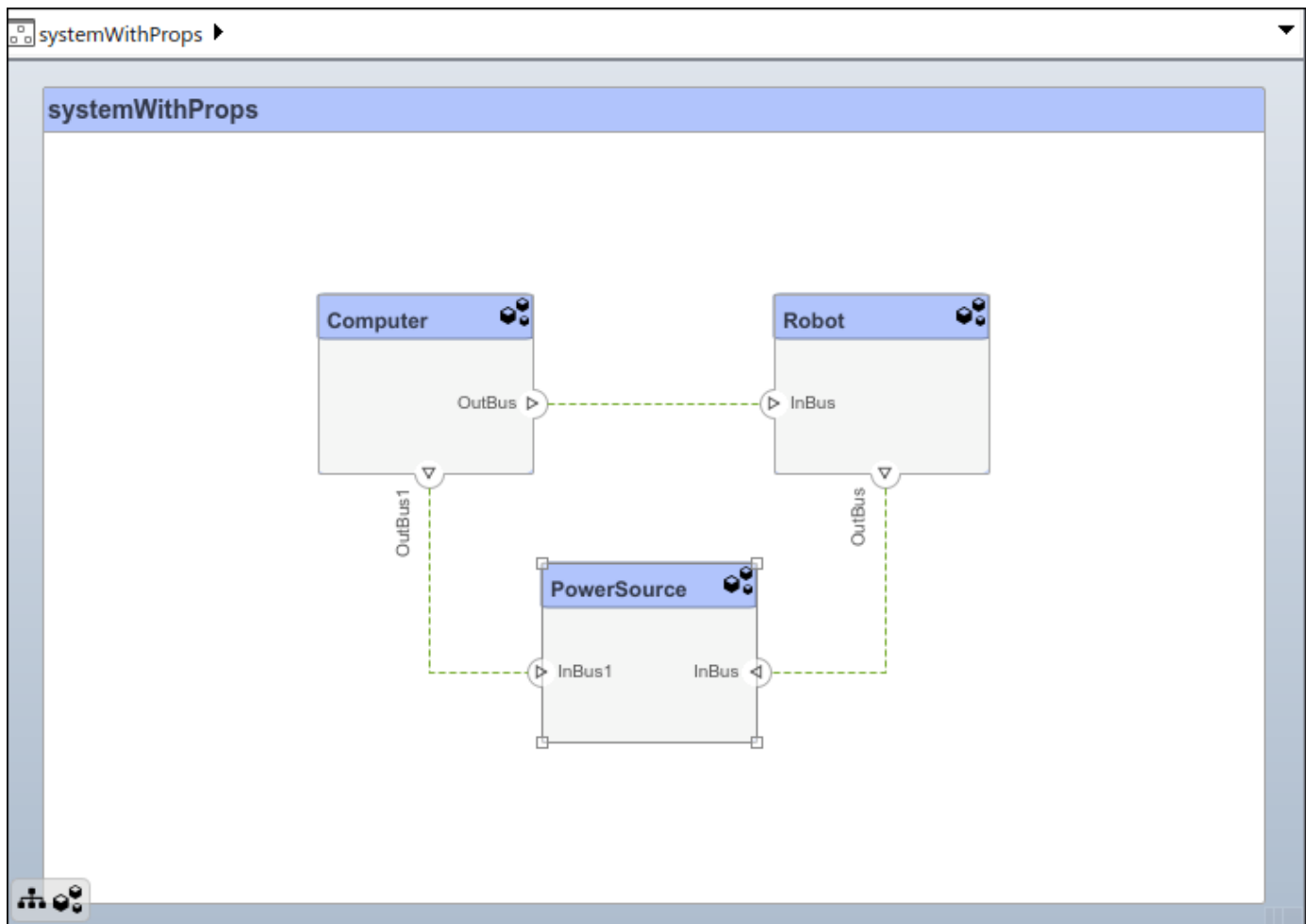
- 1 Define a profile containing a set of stereotypes that describe some analyzable properties (for example, cost and weight).
- 2 Apply the profile to an architecture model and add stereotypes from that profile to elements of the model (components, ports, or connectors).
- 3 Specify values for the properties on those elements.
- 4 Write an analysis function to compute values necessary for the trade study. This is a static constraint solver for parametrics and values of related properties captured in the system model.
- 5 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. Use the **Instantiate Architecture Model** tool.
- 6 Run the analysis function and then see analysis calculations and results in the **Analysis Viewer** tool.

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.

Open the Model

Open the systemWithProps architecture model.



Import Profile

Enable analysis of properties by first importing a profile. In the toolstrip, navigate to **Modeling > Profiles > Import** and browse to the profile to import it.

Apply Stereotypes to Model Elements

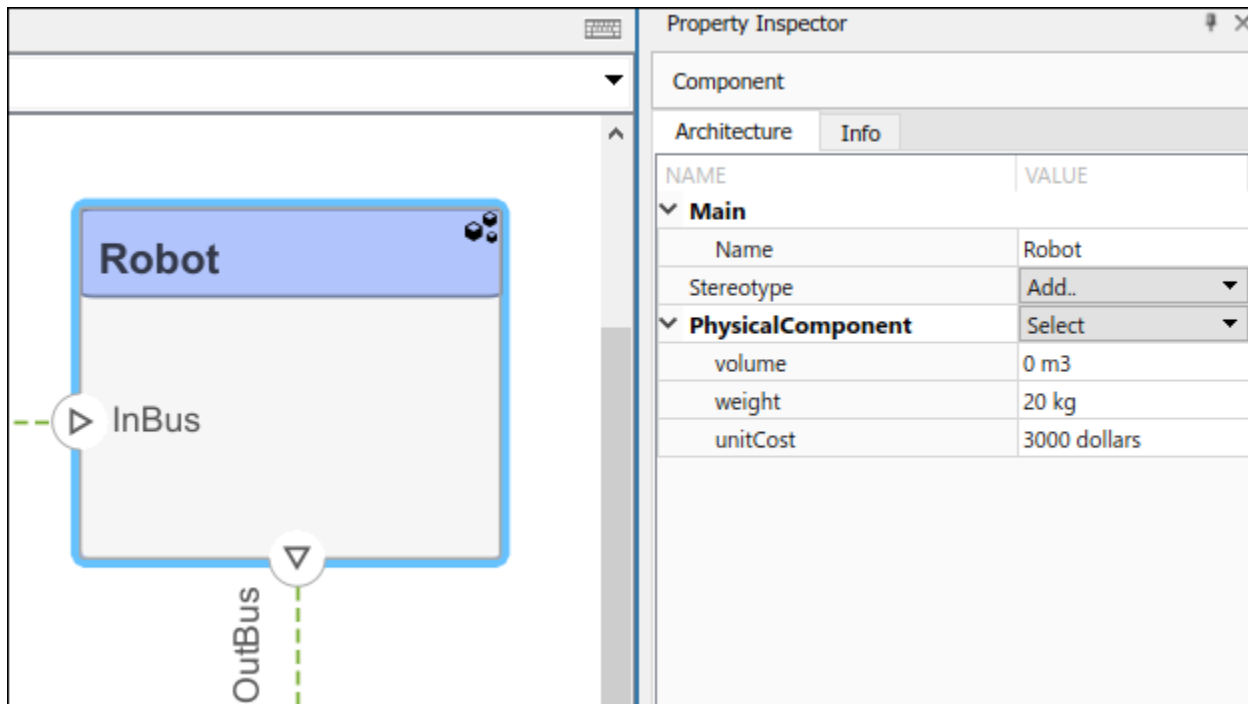
Apply stereotypes to all model elements that are part of the analysis. Use the Apply Stereotypes dialog to apply stereotypes to all elements of a certain type. Navigate to **Modeling > Apply Stereotypes**. In Apply Stereotypes, from **Apply stereotype(s) to**, select **Components**. From **Scope**, select **This layer**. For more information, see “Use Apply Stereotypes Dialog to Batch Apply Stereotypes” on page 4-15.

Tip 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 in the Property Inspector. To open the Property Inspector, navigate to **Modeling > Property Inspector**.

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



Create a Model Instance for Analysis

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

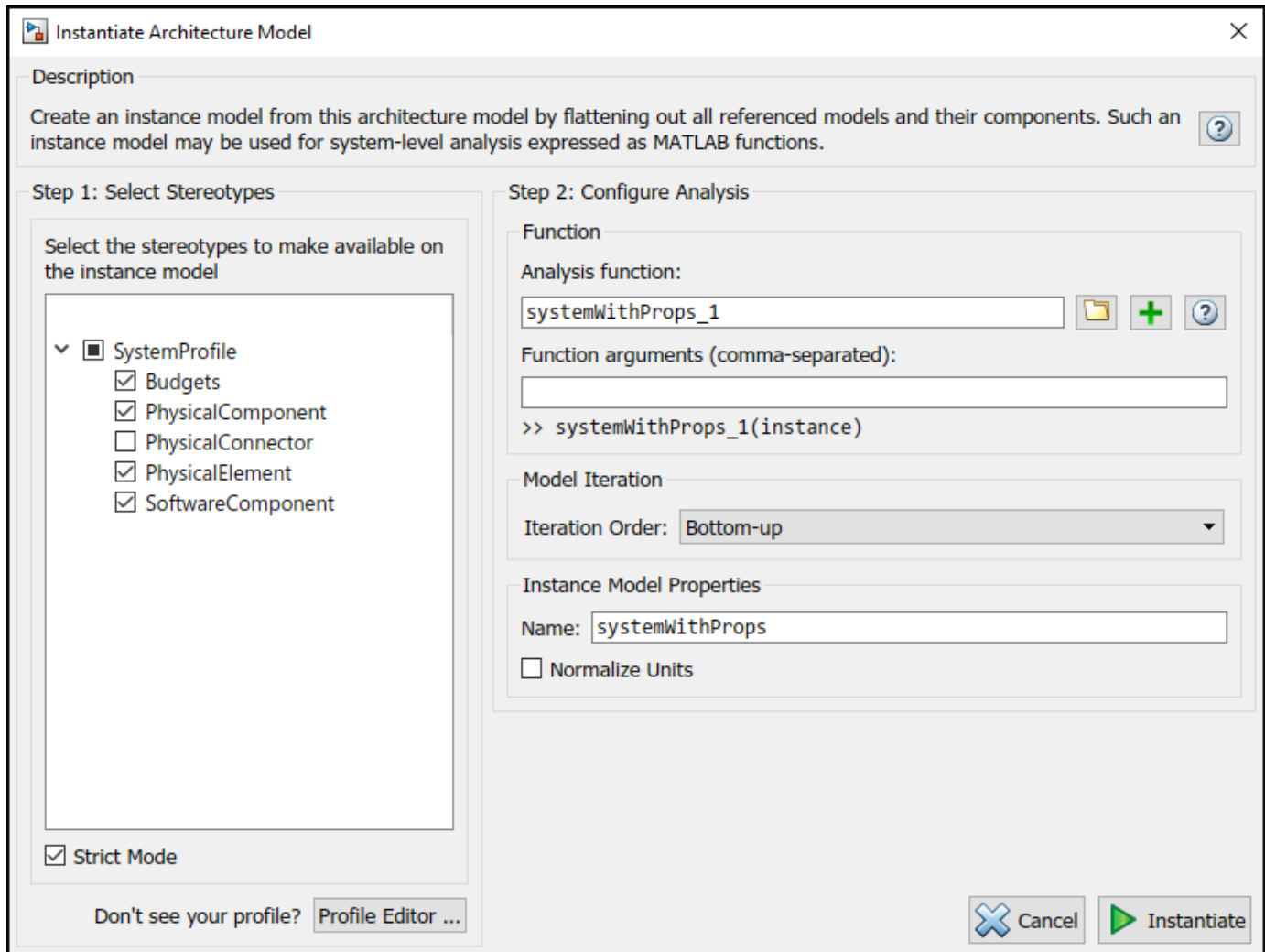
An instance is an occurrence of an architecture model element at a given point in time.

An instance freezes the active variant or model reference of the component in the instance model.

An instance model is a collection of instances.

You can update an instance model with changes to a model, but the instance model will not update with changes in active variants or model references. You can use an instance model, saved in a MAT file, of a System Composer architecture model for analysis.

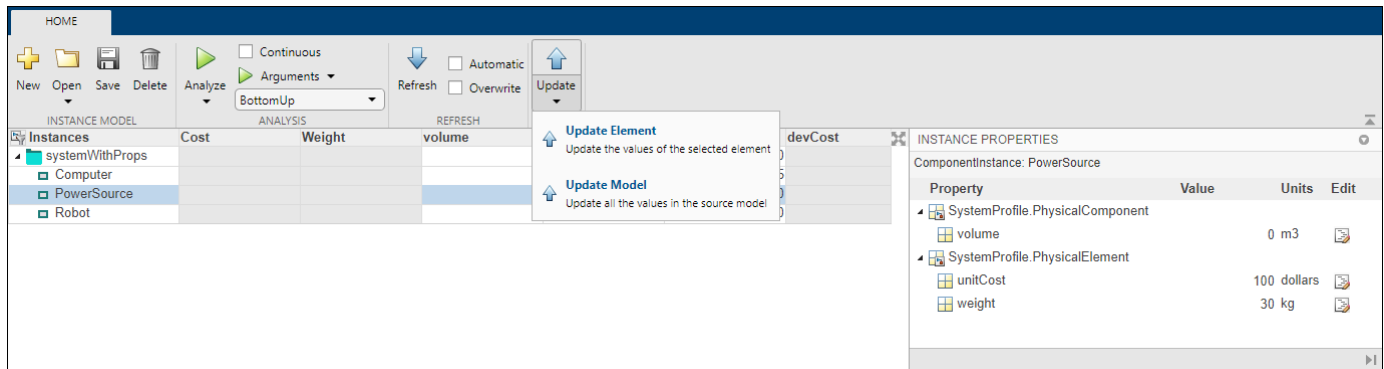
Navigate to **Modeling > Analysis Model** to open the **Instantiate Architecture Model** tool. Specify all the parameters required to create and view an analysis model.



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

To view the instance, click **Instantiate** and launch the **Analysis Viewer** tool.

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



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** pulls changes to 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 instances. An analysis function quantitatively evaluates an architecture for certain characteristics.

An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.

Use an analysis function to calculate the result of an analysis.

For more information, see “Analysis Function Constructs” on page 7-9.

You can add an analysis function as you set up the analysis instance model. 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 using the **Analysis Viewer** tool.

```
function systemWithProps_1(instance,varargin)

if instance.isComponent() && ~isempty(instance.Components)...
    && instance.hasValue('SystemProfile.PhysicalElement.unitCost')
    sysComponent_unitPrice = 0;
    for child = instance.Components
        if child.hasValue('SystemProfile.PhysicalElement.unitCost')
            comp_price = child.getValue('SystemProfile.PhysicalElement.unitCost');
            sysComponent_unitPrice = sysComponent_unitPrice + comp_price;
        end
    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() && ~isempty(instance.Components)...
    && instance.hasValue('SystemProfile.PhysicalElement.weight')
    weight = 0;
    for child = instance.Components
        if child.hasValue('SystemProfile.PhysicalElement.weight')
            subcomp_weight = child.getValue('SystemProfile.PhysicalElement.weight');
            weight = weight + subcomp_weight;
        end
    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, and process the subcomponents of that component recursively before moving to a sibling component.
 - **Top-Down** — Like pre-order, but process all sibling components before moving to their subcomponents.
 - **Post-order** — Start from components with no subcomponents, process each sibling, and then move to parent.
 - **Bottom-up** — Like post-order, but process all subcomponents at the same depth before moving to their parents.

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 highlighted yellow in the **Analysis Viewer**.

Instances	Cost	Weight	volume	unitCost	weight	devCost
systemWithProps			0	5100	55	
Computer			0	2000	5	
PowerSource			0	100	30	
Robot			0	3000	20	

Here, the total cost of the system is 5100 dollars and the total weight is 55 kg.

See Also

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

More About

- “Define Profiles and Stereotypes” on page 4-2
- “Organize System Composer Files in Projects” on page 10-2
- “Analysis Function Constructs” on page 7-9
- “Calculate Endurance Using Quadcopter Architectural Design” on page 7-16
- “Battery Sizing and Automotive Electrical System Analysis” on page 7-14

Analysis Function Constructs

Analyze architectures to choose between design alternatives or improve existing designs. You can use analysis functions with System Composer architecture models to perform systems analysis and trade studies.

An analysis function is a MATLAB function that computes values necessary to evaluate the architecture using the properties of each element in the model instance.

Use an analysis function to calculate the result of an analysis and determine the optimal parameters to use for behavior models to simulate the architectural system.

Type	Section
Roll-up analysis	“Roll-Up Analysis for Quadcopter Design” on page 7-9
Class-based analysis	“Class-Based Analysis for Battery Sizing” on page 7-10
Allocation-based analysis	“Allocation-Based Analysis for Tire Pressure Monitoring” on page 7-11
Remaining useful life (RUL) analysis	“Remaining Useful Life Analysis for Mobile Robot Design” on page 7-11
Variant analysis	“Variant Analysis for Insulin Infusion Pump Design” on page 7-12

For more information on analysis functions and architecture instances, see “Analyze Architecture” on page 7-2.

Roll-Up Analysis for Quadcopter Design

Use a roll-up analysis function to calculate a total or average of model element property values. Assign properties to model elements using stereotypes. For more information, see “Define Profiles and Stereotypes” on page 4-2.

In this example, the analysis function `systemWithProps_1` calculates the total cost of all components in the model and is compatible with the **Analysis Viewer** tool.

```
function systemWithProps_1(instance,varargin)
if instance.isComponent() && ~isempty(instance.Components)...
    && instance.hasValue('SystemProfile.PhysicalElement.unitCost')
        sysComponent_unitPrice = 0;
        for child = instance.Components
            if child.hasValue('SystemProfile.PhysicalElement.unitCost')
                comp_price = child.getValue('SystemProfile.PhysicalElement.unitCost');
                sysComponent_unitPrice = sysComponent_unitPrice + comp_price;
            end
        end
        instance.setValue('SystemProfile.PhysicalElement.unitCost',sysComponent_unitPrice);
end
```

This analysis function iterates through an architecture instance. First, the `sysComponent_unitPrice` variable is set to zero so that every time the analysis is run, sums do not accumulate indefinitely. Each component instance is checked for a `unitCost` property value. All `unitCost` property values are summed up and saved in the `sysComponent_unitPrice` variable. Finally, the `unitCost` property of the current component instance is updated with the value of `sysComponent_unitPrice`. For more information, see “Write Analysis Function” on page 7-6.

In this example, a section of the analysis function `calculateEndurance` calculates endurance for a quadcopter using component instance properties. The calculated endurance value is then set for the architecture instance of the quadcopter with the `setValue` function.

```

if payloadBatteryCapacity == 0
    totalPower = powerConsumption + hoverPower/efficiency;
    endurance = (batteryCapacity/1000)/(totalPower/voltage)*60;
else
    payloadEndurance = (payloadBatteryCapacity/1000)/(powerConsumption/voltage)*60;
    flightEndurance = (batteryCapacity/1000)/((hoverPower/efficiency)/voltage)*60;
    if flightEndurance < payloadEndurance
        endurance = flightEndurance;
    else
        endurance = payloadEndurance;
        warning('Endurance is limited by payload electronics.')
    end
end
instance.setValue('AirVehicle.Endurance',endurance)

```

For more information and for the supporting files, see “Calculate Endurance Using Quadcopter Architectural Design” on page 7-16.

Class-Based Analysis for Battery Sizing

Use MATLAB classes for an analysis function to iterate over an object, or instantiation of the class.

In this example, the class called `computeBatterySizing` involves properties and methods useful for the analysis function `computeLoad`.

```

classdef computeBatterySizing < handle

    properties
        totalCrankingInrushCurrent;
        totalCrankingCurrent;
        totalAccessoriesCurrent;
        totalKeyOffLoad;
        batteryCCA;
        batteryCapacity;
        puekertcoefficient;
    end

    methods
        function obj = computeBatterySizing(obj)
            obj.totalCrankingInrushCurrent = 0;
            obj.totalCrankingCurrent = 0;
            obj.totalAccessoriesCurrent = 0;
            obj.totalKeyOffLoad = 0;
            obj.batteryCCA = 0;
            obj.batteryCapacity = 0;
            obj.puekertcoefficient = 1.2;
        end

        function obj = displayResults(obj)
            tempNumdaysToDischarge = ((obj.batteryCapacity/obj.puekertcoefficient)*0.3)/(obj.totalKeyOffLoad*1e-3)/24;
            disp("Total KeyOffLoad: " + num2str(obj.totalKeyOffLoad) + " mA");
            disp("Number of days required for KeyOffLoad to discharge 30% of battery: " + ...
                num2str(tempNumdaysToDischarge) + ".");
            disp("Total CrankingInRush current: " + num2str(obj.totalCrankingInrushCurrent) + " A");
            disp("Total Cranking current: " + num2str(obj.totalCrankingCurrent) + " A");

            if(obj.totalCrankingCurrent > obj.batteryCCA)
                disp("The Cold Cranking Amps of the specified battery is not sufficient to start the car @ F.")
            else
                disp("CCA of the specified battery is sufficient to start the car at @ F.")
            end
        end
    end
end

```

For more information and for the supporting files, see “Battery Sizing and Automotive Electrical System Analysis” on page 7-14.

Allocation-Based Analysis for Tire Pressure Monitoring

A functional-to-logical allocation matrix allocates components in a functional architecture to components in a logical architecture. Coverage analysis is the most basic form of analysis to determine whether all elements have been allocated.

First, open the project for this example. Then, load the allocation set and collect the scenarios.

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

Verify that each function in the system is allocated.

```
import systemcomposer.query.*;
[~, allFunctions] = allocSet.SourceModel.find(HasStereotype(IsStereotypeDerivedFrom("TPMSProfile.Function")));
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 output verifies that all functions are allocated.

For more information and for the supporting files, see “Allocate Architectures in Tire Pressure Monitoring System” on page 6-5.

Remaining Useful Life Analysis for Mobile Robot Design

Remaining useful life (RUL) analysis estimates the time remaining before different subsystems fail. The goal is to anticipate maintenance and thus minimize system disruptions.

In this example, the analysis function `scMobileRobotAnalysis` is compatible with the **Analysis Viewer** tool.

```
function scMobileRobotAnalysis(instance,varargin)
    ExpectedYearsBeforeFirstMaintenance = 2;

    if ~instance.isArchitecture()
        if instance.hasValue("HardwareBaseStereotype.Life")
            Life = instance.getValue("HardwareBaseStereotype.Life");
            UsagePerDay = instance.getValue("HardwareBaseStereotype.UsagePerDay");
            UsagePerYear = instance.getValue("HardwareBaseStereotype.UsagePerYear");
            WillSurvive = Life > UsagePerDay * UsagePerYear * ExpectedYearsBeforeFirstMaintenance;
            instance.setValue("HardwareBaseStereotype.ExceedExpectedMaintenance", WillSurvive);
        end
    end
end
```

After running this analysis function, you can optimize the desired first expected maintenance time in years. Each component that exceeds the expected maintenance time, in this case set to two years, is flagged with a check box. Unchecked components should be optimized or replaced with longer-lasting parts.

For more information and for the supporting files, see “Define Stereotypes and Perform Analysis” on page 4-34.

Variant Analysis for Insulin Infusion Pump Design

Use variant analysis to choose one optimal combination of variants by comparing them with a calculated metric.

In this example, the analysis function `OutcomeAnalysis` is used to determine the best configuration for an insulin infusion pump. This standalone analysis function does not involve the **Analysis Viewer** tool. Instead, the analysis function uses the `iterate` function and can be executed directly from the MATLAB Command Window.

The `OutcomeAnalysis` function first gathers all variant choice components named `Pump` and `BGSensor`.

```
function outcomes = OutcomeAnalysis()

modelName = 'InsulinInfusionPumpSystem';

therapyModel = systemcomposer.openModel(modelName);
components = therapyModel.Architecture.Components;
for idx = 1:numel(components)
    if strcmp(components(idx).Name, 'Pump')
        pumps = components(idx).getChoices;
        pumpNames = {};
        for jdx = 1:numel(pumps)
            pumpNames{end+1} = pumps(jdx).Name;
        end
    elseif strcmp(components(idx).Name, 'BGSensor')
        sensors = components(idx).getChoices;
        sensorNames = {};
        for jdx = 1:numel(sensors)
            sensorNames{end+1} = sensors(jdx).Name;
        end
    end
end
end
```

The analysis function then collects all variant combinations to iterate over.

```
config.Sensor = sensorNames{1};
config.Pump = pumpNames{1};
configs = {};

for idx = 1:numel(sensorNames)
    for jdx = 1:numel(pumpNames)
        config.Sensor = sensorNames{idx};
        config.Pump = pumpNames{jdx};
        configs{end+1} = config;
    end
end
```

The analysis function activates the variants one by one, iterates over the model properties, and collects outcomes. To set variant combinations, `OutcomeAnalysis` uses the `setVariants` function. To compute the outcomes, `OutcomeAnalysis` uses the `computeOutcome` function.

```

outcomes = {};

for idx = 1:numel(configs)
    hOutcome = OutcomeContainer(configs{idx});
    therapyModel.iterate('Topdown',@setVariants,configs{idx});
    therapyModel.iterate('BottomUp',@computeOutcome,hOutcome);
    hOutcome.setWeights([1e-6 1 10 1 1000]);
    outcomes{end+1} = hOutcome;
end

```

Finally, the analysis function plots the net outcome to reveal the optimal design choice.

```

properties = {'Lower NRE','Higher Accuracy','Better Compliance',...
'Sooner To Market','Lower Operating Cost'};
plotMatrix = zeros(numel(outcomes), numel(properties));
plotStrings = {};
for idx = 1:numel(outcomes)
    plotStrings{idx} = [outcomes{idx}.Sensor '+' outcomes{idx}.Pump];
    plotMatrix(idx,1) = 1/(outcomes{idx}.NRE);
    plotMatrix(idx,2) = outcomes{idx}.Accuracy;
    plotMatrix(idx,3) = outcomes{idx}.Compliance;
    plotMatrix(idx,4) = 1/(outcomes{idx}.TimeToMarket);
    plotMatrix(idx,5) = 1/(outcomes{idx}.AnnualCost);
end

colmin = zeros(1,5);
colmax = max(plotMatrix);
normalizedMatrix = rescale(plotMatrix, 'InputMin', colmin, 'InputMax', colmax);

if exist('spider_plot') == 2
    fig = figure;
    spider_plot(normalizedMatrix,'AxesLabels',properties,'FillOption','on',...
        'FillTransparency',0.1,'AxesDisplay','one');

    title(sprintf('Trade Study Outcome'),...
        'FontSize', 14);
    legend(plotStrings, 'Location', 'eastoutside');
    pos = fig.Position;
    pos(2) = pos(2) - pos(4);
    pos(3) = 2*pos(3);
    pos(4) = 2*pos(4);
    fig.Position = pos;
else
    vals = sum(normalizedMatrix,2)/5;
    x_labels = categorical(plotStrings);
    h = bar(x_labels,vals);
    title('Net outcome');
    ax = h.Parent;
    ax.YLabel.String = 'Normalized units';
end

```

For more information and for the supporting files, see “Design Insulin Infusion Pump Using Model-Based Systems Engineering” on page 7-23.

See Also

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

More About

- “Define Profiles and Stereotypes” on page 4-2
- “Analyze Architecture” on page 7-2
- “Organize System Composer Files in Projects” on page 10-2

Battery Sizing and Automotive Electrical System Analysis

Overview

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

Structure of Model

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

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

Load Model and Run Analysis

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

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

```
objcomputeBatterySizing = computeBatterySizing;
```

Run the analysis using the iterator.

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

Display analysis results.

```
objcomputeBatterySizing.displayResults
```

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

```
ans =
  computeBatterySizing with properties:

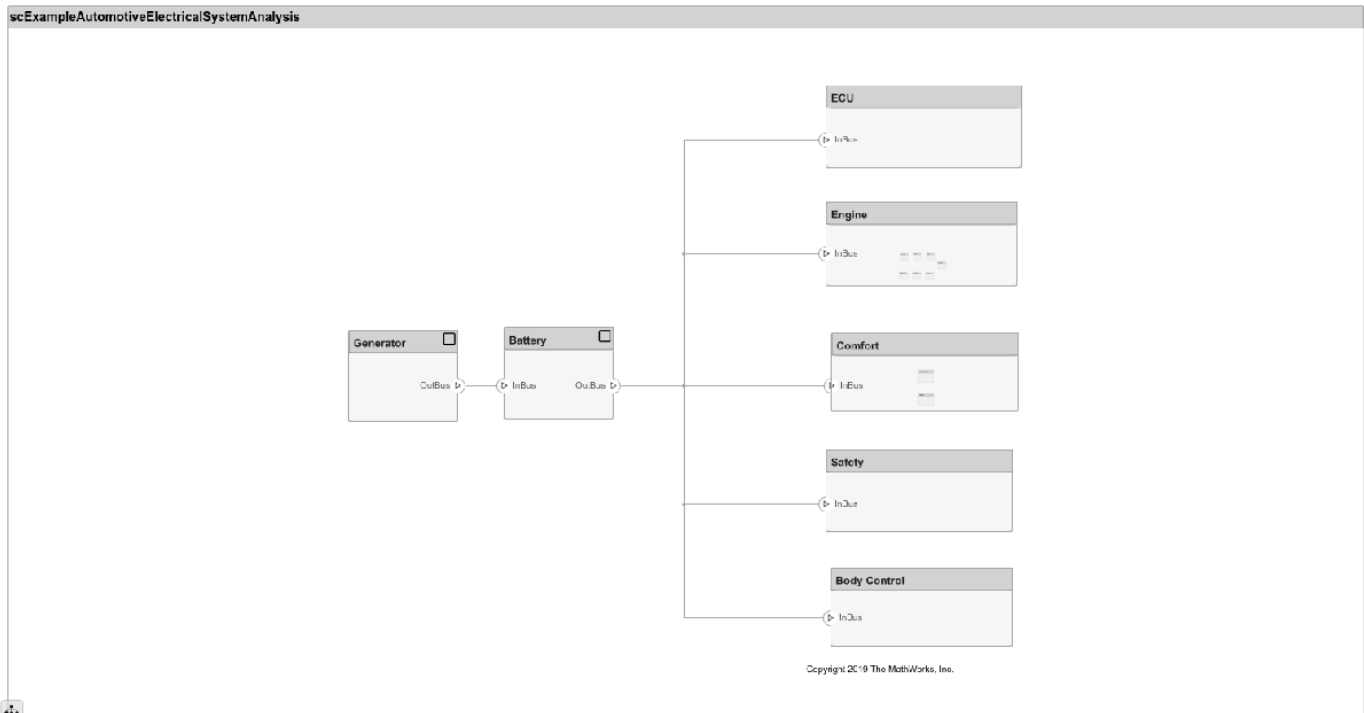
    totalCrankingInrushCurrent: 70
      totalCrankingCurrent: 104
    totalAccesoriesCurrent: 71.6667
      totalKeyOffLoad: 158.7080
```



```

batteryCCA: 500
batteryCapacity: 850
puekertcoefficient: 1.2000

```



Close Model

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

See Also

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

More About

- “Analyze Architecture” on page 7-2
- “Analysis Function Constructs” on page 7-9
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22
- “Allocate Architectures in Tire Pressure Monitoring System” on page 6-5
- “Calculate Endurance Using Quadcopter Architectural Design” on page 7-16

Calculate Endurance Using Quadcopter Architectural Design

This example shows you how to create the physical architecture of a quadcopter following a target green ball using System Composer™ and Requirements Toolbox™ and following a model-based systems engineering (MBSE) workflow. Start by defining requirements, then extend architectural data using stereotypes and custom property values for model elements, and finally use analysis to iteratively improve on the design.

Define Functional Requirements for Quadcopter Design

The first step in the MBSE methodology is to define requirements. The concept of operations, or *conops*, define the overall idea of the system. You then derive functional requirements from conops requirements and further define the logical and physical subsystems by linking requirements.

1. Load Simulink® customizations.

```
sl_refresh_customizations
```

2. Load the physical architecture model in memory to view its requirement links.

```
systemcomposer.loadModel("QuadArchPhysical");
```

3. Open the requirement sets.

- Concept of operations
- Functional requirements
- Logical requirements
- Physical requirements

```
slreq.open("conops");  
slreq.open("FunctionalReqs_Quad");  
slreq.open("LogicalReqs_Quad");  
slreq.open("PhysicalReqs_Quad");
```

4. Open the Requirements Editor (Requirements Toolbox).

```
slreq.editor
```

Inspect the conops requirement **Target Characteristics**. The requirements under the **Decomposed by** list represent the requirements contained in the top-level requirement. The requirement **Target Identification** under the **Derived from** list represents requirements derived from the conops requirement.

Index	ID	Summary
conops		
> 1	#1	Target Characteristics
> 2	#7	System Requirements
FunctionalReqs_Quad		
> 1	#1	Target Identification
> 2	#6	System Mode Management
> 3	#7	System Flight Management
> 4	#12	System Flight Control
LogicalReqs_Quad		
> 1	#10	Ground Control Station Requir
> 2	#20	Air Vehicle
> 3	#30	Flight Control System
> 4	#35	Payload
PhysicalReqs_Quad		
> 1	#3	Power System
> 2	#6	Mass Properties & Geometry
> 3	#9	Ground Control Station
> 4	#10	Quadcopter
> 5	#13	Payload

Requirement: #1

Details

Properties

Type: Functional

Index: 1

Custom ID: #1

Summary: Target Identification

Description Rationale

The system shall be able to identify the target's position relative to the system.

Keywords:

Revision information:

Links

⇒ Decomposed by:

- System Video Capture
- Identify Target Direction
- Identify Target Distance
- Identify Target Location

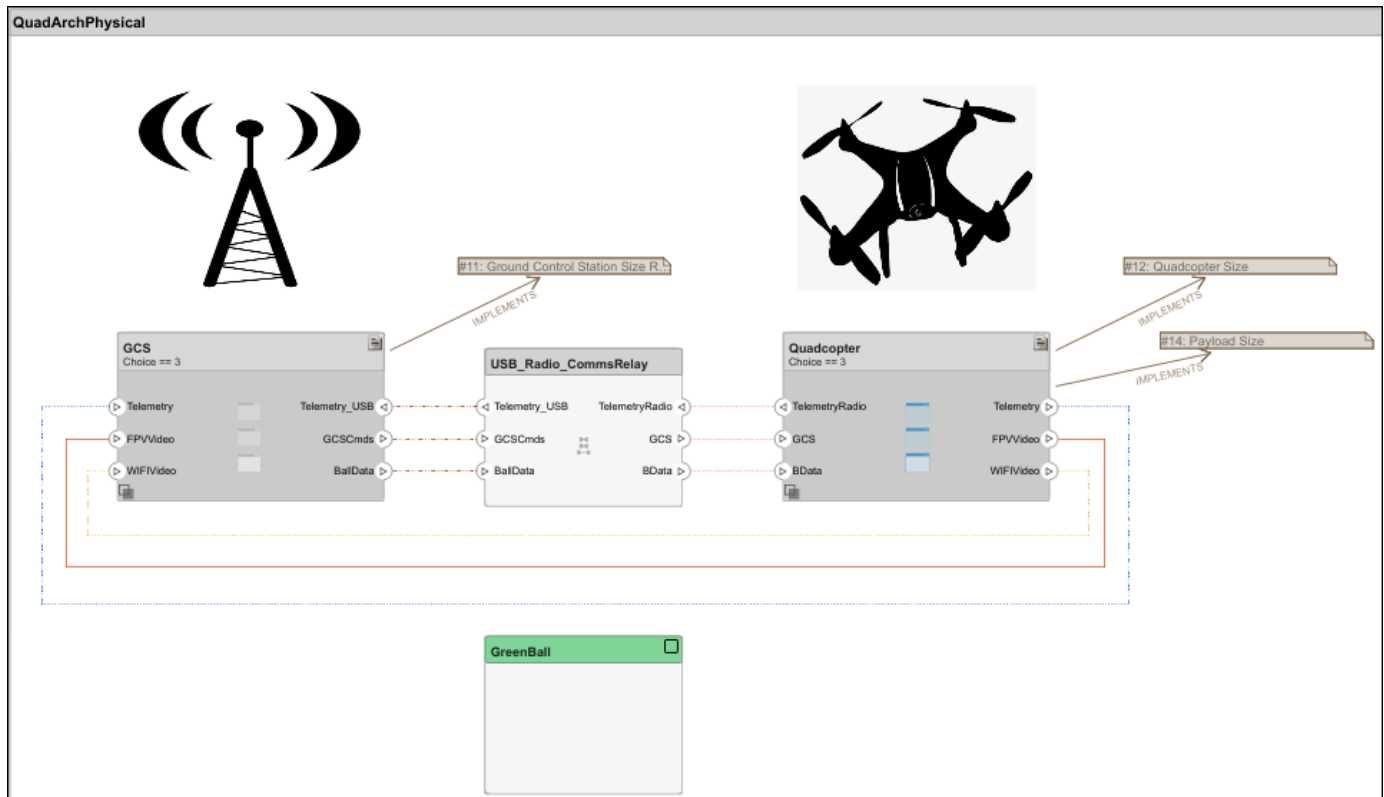
⇒ Derives:

- Target Characteristics

To open the quadcopter physical architecture model, run this code.

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

Manage requirements and architecture together in the **Requirements Manager** from Requirements Toolbox. Navigate to **Apps > Requirements Manager**. You are now in the Requirements perspective in System Composer. In this perspective, you can see which requirements are associated with specific components in the physical architecture.



Specify Functional Design Using Stereotypes and Properties

Stereotypes, defined on a profile, include properties to specify metadata on model elements to which stereotypes are applied.

To open the Profile Editor tool, on the System Composer toolstrip, navigate to **Modeling > Profile Editor**. Alternatively, run this command.

```
systemcomposer.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 X Stereotype New Stereotype X Import into Select ?

Profile Browser
Filter profiles: <all>

- ▼ QuadcopterComms
 - BLE
 - CSI_Camera
 - CommsChannel
 - FPV_Camera
 - Radio
 - Serial_SPI
 - USB
 - Wifi
 - Wifi_Camera
- ▼ QuadcopterPhysicalProperties
 - AirVehicle
 - Battery
 - HW_Implementation
 - Payload
 - PayloadComponent
 - Target

Stereotype Properties
Name: AirVehicle
Applies to: Component Icon

Base stereotype: QuadcopterPhysicalProperties.HW_Implementation

Abstract stereotype

Description:

► Default Stereotypes for Composition

	Property name	Type	Name	Unit	Default
1	PayloadCapacity	double	n/a	g	0
2	BatteryCapacity	double	n/a	mAh	0
3	Cost	double	n/a	\$	0
4	FullyReprogrammable	boolean	n/a	n/a	<input type="checkbox"/>
5	IntegratedFwdCamera	boolean	n/a	n/a	<input type="checkbox"/>
6	PayloadPerDollar	double	n/a	g/\$	0
7	Endurance	double	n/a	min	0
8	AirframeMass	double	n/a	g	0
9	TotalMass	double	n/a	g	0
10	PowerDraw	double	n/a	W	0
11	PayloadBatteryCapa...	double	n/a	mAh	0

Show inherited properties (read-only)


The **AirVehicle** stereotype applies to components and inherits from the base stereotype **HW_Implementation**. Each property under the **AirVehicle** stereotype is specified by a data type defined by **Type**, and some properties include an engineering unit defined by **Unit**. You can apply the **AirVehicle** stereotype to components in the quadcopter physical architecture to elaborate on these components with specific property values. Define these property values for the **RPiCam_RadioComms** component in the Property Inspector.

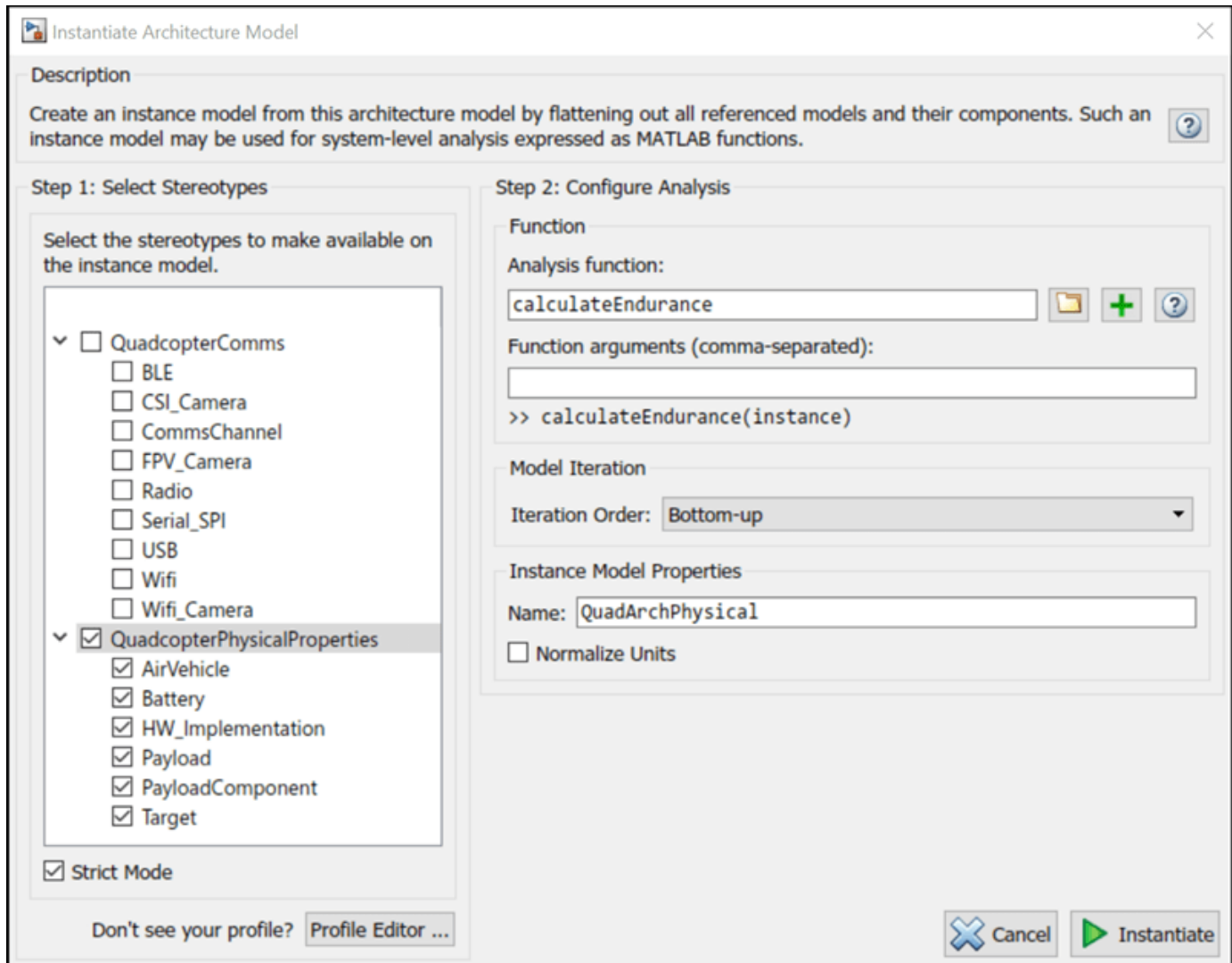
The screenshot displays the System Composer interface. The main workspace shows an architecture model for 'RPiCam_RadioComms' within a 'Quadcopter' system. The model includes components like 'Battery', 'OnboardProcessor', and 'Payload', connected by dashed lines. The 'Property Inspector' on the right shows the following details:

NAME	VALUE
Main	
Name	RPiCam_RadioComms
Stereotype	Add..
AirVehicle	
PayloadCapacity	114 g
BatteryCapacity	0 mAh
Cost	0 \$
<input checked="" type="checkbox"/> FullyReprogrammable	
<input type="checkbox"/> IntegratedFwdCamera	
PayloadPerDollar	0 g/\$
Endurance	0 min
AirframeMass	46 g
TotalMass	0 g
PowerDraw	0 W
PayloadBatteryCapacity	0 mAh
HW_Implementation	PhysicalDevice

Perform Roll-Up Analysis to Calculate Endurance for Quadcopter Design

To open the Instantiate Architecture Model tool, on the System Composer toolstrip, navigate to **Modeling > Analysis Model**. Select all the stereotypes under the

QuadcopterPhysicalProperties profile. Click the open  button, then open the analysis function file calculateEndurance.m. Select Bottom-up for **Iteration Order**. Click **Instantiate**.



In the Analysis Viewer tool, you can use an analysis function to calculate roll-up property values such as `BatteryCapacity`, `PayloadBatteryCapacity`, `PowerDraw`, and `TotalMass`. The analysis function also calculates the performance characteristics `PowerDraw` and `Endurance`. For more information, see “Analysis Function Constructs” on page 7-9. Click **Analyze** to view the analysis results highlighted in yellow.

Instances	AirframeMass	BatteryCapacity	Cost	Endurance	
<ul style="list-style-type: none"> ▾ QuadArchPhysical <ul style="list-style-type: none"> ▣ GreenBall ▾ Quadcopter <ul style="list-style-type: none"> ▣ Battery ▾ Payload <ul style="list-style-type: none"> ▣ Camera ▣ PayloadBattery ▣ PayloadPowerSwitch ▣ VideoProcessing 		46	650	0	3.824743583

The Endurance property for this particular configuration is calculated as approximately 3.825 using this equation.

$$endurance = \frac{\left(\frac{batteryCapacity}{1000}\right)}{\left(\frac{totalPower}{voltage}\right)} * 60$$

You can change the variant configuration and run the analysis function again to calculate Endurance and compare different proposed designs.

See Also

`systemcomposer.profile.editor` | `slreq.editor` | `sl_refresh_customizations`

More About

- “Analyze Architecture” on page 7-2
- “Compose Architecture Visually” on page 1-2
- “Analysis Function Constructs” on page 7-9
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22
- “Modeling System Architecture of Keyless Entry System” on page 9-25

Design Insulin Infusion Pump Using Model-Based Systems Engineering

This example shows you how to use a model-based systems engineering workflow to investigate optimal insulin infusion pump design. Insulin pumps are medical devices used by diabetic humans that mimic the human pancreas by delivering insulin continuously and delivering variable amounts of insulin with food intake.

The purpose of an insulin pump wearable device is to keep the blood glucose level of the wearer near a healthy set point by infusing insulin as needed and in response to food intake. This example shows a proposed insulin infusion pump system with two sensor and three pump variants that represent alternate design choices.

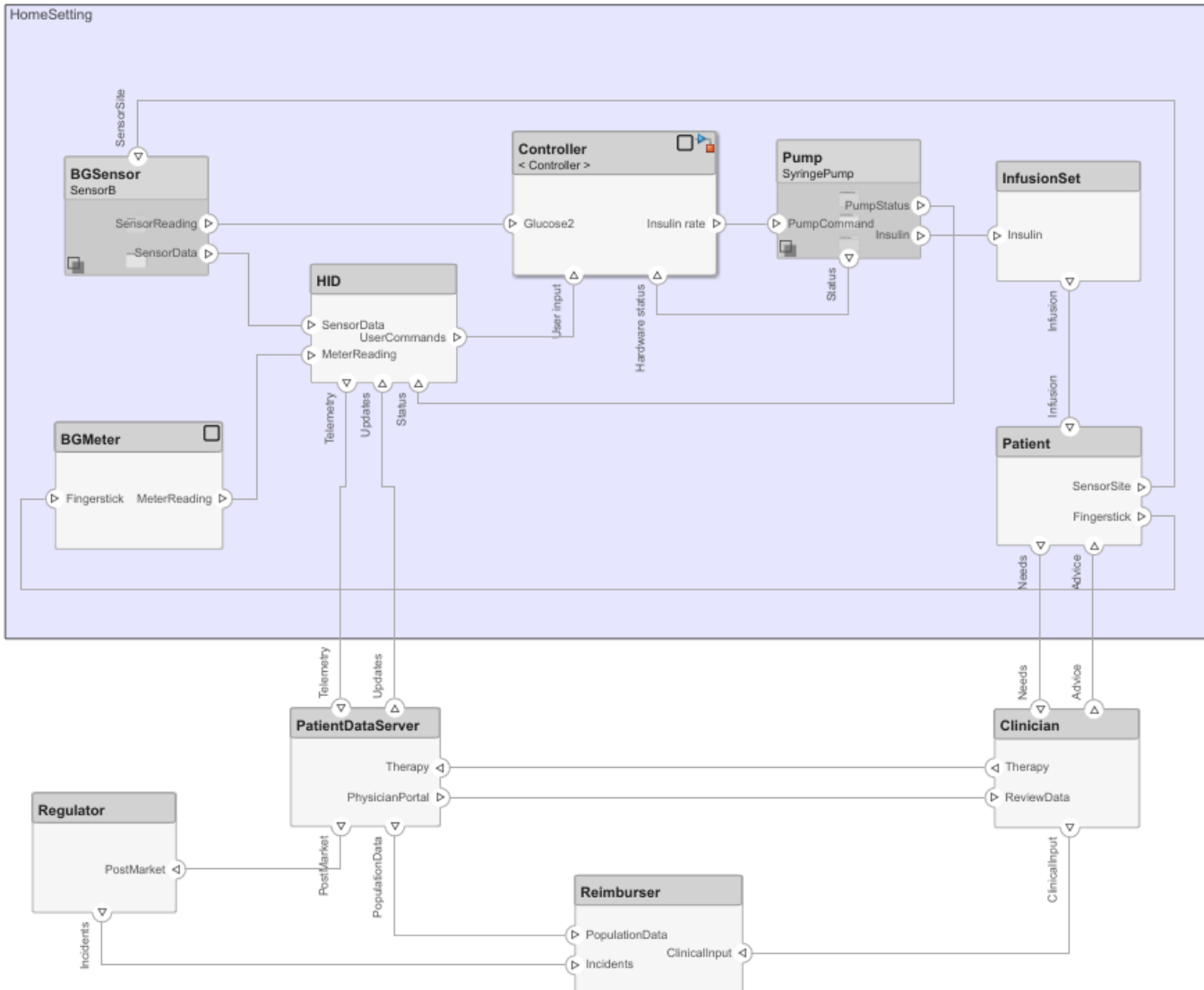
Begin by determining system requirements, then create detailed design models with code generation and verification tests. Finally, simulate the system architecture model that meets the evolving requirements.

Insulin Pump System Architecture Model

This figure shows the System Composer™ architecture model for the insulin pump system. This example uses Stateflow® blocks. If you do not have a Stateflow license, you can open and simulate the model but can only make basic changes, such as modifying block parameters.

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

The `BGSensor` component measures the blood glucose level. The `Controller` component makes a decision about insulin rate. The `Pump` component provides insulin to the body using the `InfusionSet`. The `Patient` receives the treatment. The `BGMeter` calibrates the `BGSensor`. Finally, the `HID` (human interface device) component may be a mobile app on the phone for the patient to communicate with the system. The `HID` provides information to the `PatientDataServer` component, which sends analyses to the `Clinician`, `Regulator`, and `Reimbursor` components.



System Requirements and Links

Use Requirements Toolbox™ to identify the system requirements, further break them down into subsystem requirements, and link derived requirements to architectural components that satisfy them. A Requirements Toolbox license is required to link, trace, and manage requirements in System Composer.

Manage requirements and architecture together in the Requirements Perspective from Requirements Toolbox. Select **Apps > Requirements Manager**. To edit requirements, select **Requirements > Requirements Editor** or enter these commands to open the Requirements Editor (Requirements Toolbox).

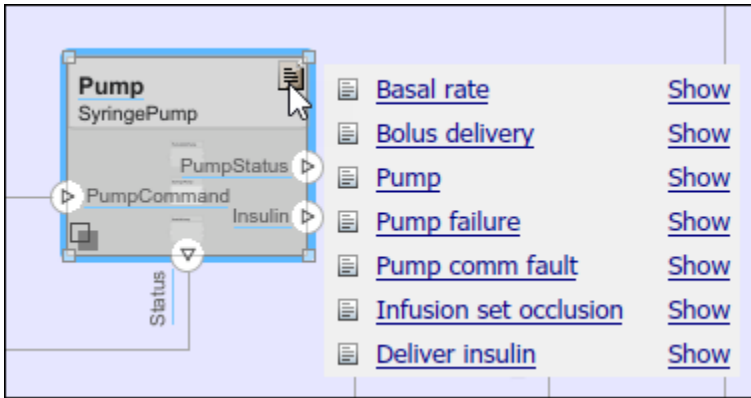
```
slreq.open("Infusion_Pump_System");
slreq.open("Insulin_Pump_Controller_Software_Specification");
slreq.editor
```

Index	ID	Summary
▼		Infusion_Pump_System
>	1	#1 Sense blood glucose
>	2	#3 Status view
▼	3	#7 Deliver insulin
		#12 Basal rate
		#13 Bolus delivery
		#14 Override
▼	4	#8 Alarm monitor
		#15 Sensor needs calibration
		#16 Sensor life expiring
>		#17 Low battery
>		#23 Communication faults
		#27 Running out of medication
>		#28 Pump failure
	5	#10 Remote patient monitoring
	6	#11 Population monitoring
▼		Insulin_Pump_Controller_Software_Specification
	1	#1 Basic Theory of Operation
▼	2	#2 Functional Requirements
>		#3 Startup
>		#8 Basal infusion
>		#9 Bolus infusion
>		#24 Alarm handling

The requirements decomposition and analysis at this point represent these concerns:

- Accuracy of delivery
- Mitigations against over-infusion, which leads to dangerously low blood glucose levels
- Fault analysis to prevent negative outcomes, for example, when the battery dies or the device runs out of medication

On the architecture model, select the requirements icon to see the requirements that are associated with the component. For example, below are the requirements linked to the Pump component.



Conversely, select a requirement to see the highlighted component by which the requirement is implemented. For example, the BGSensor component implements the Sense blood glucose requirement.

The screenshot shows the 'InsulinInfusionPumpSystem' architecture tool. The main window displays a 'BGSensor' component (SensorB) with interfaces 'SensorReading' and 'SensorData'. The 'Requirements - InsulinInfusionPumpSystem' panel is open, showing a table of requirements. Requirement #1, 'Sense blood glucose', is selected and highlighted. The 'Property Inspector' on the right shows the details for this requirement.

Index	ID	Summary
1	#1	Sense blood glucose
2	#3	Status view
3	#7	Deliver insulin
4	#8	Alarm monitor
5	#10	Remote patient monitoring
6	#11	Population monitoring

Property Inspector: Requirement: #1

Details

Properties

- Type: Functional
- Index: 1
- Custom ID: #1
- Summary: Sense blood glucose

Description

Blood glucose shall be periodically measured and provided to other system components

Links

Implemented by: BGSensor

Outcome Analysis for Optimal Design Choice

Outcome analysis consists of a trade study where the goal is to maximize a compliance score based on different metrics in the system. The compliance is a function of other properties that represent the burden to a user versus the benefits. The compliance score includes these considerations.

- Energy consumption
- Size and weight
- Accuracy
- Non-recurring engineering cost (NRE)
- Mean time between failures (MTBF)
- Consumable cost
- Ease of use

Navigate to **Modeling > Profiles > Profile Editor**, or enter this command.

```
systemcomposer.profile.editor
```

A System Composer profile, defined in the Profile Editor, is composed of stereotypes with properties defined. You can apply stereotypes to components in the model to assign specific property values to each component.

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 X Stereotype New Stereotype X Import into Select ?

Profile Browser
Filter profiles: <all>

- InsulinSystemProfile
 - Arch
 - BGMeter
 - DurableComponent
 - Pump
 - Sensor

Stereotype Properties
Name: DurableComponent
Applies to: Component Icon
Base stereotype: <nothing>
 Abstract stereotype
Description:

► Default Stereotypes for Composition

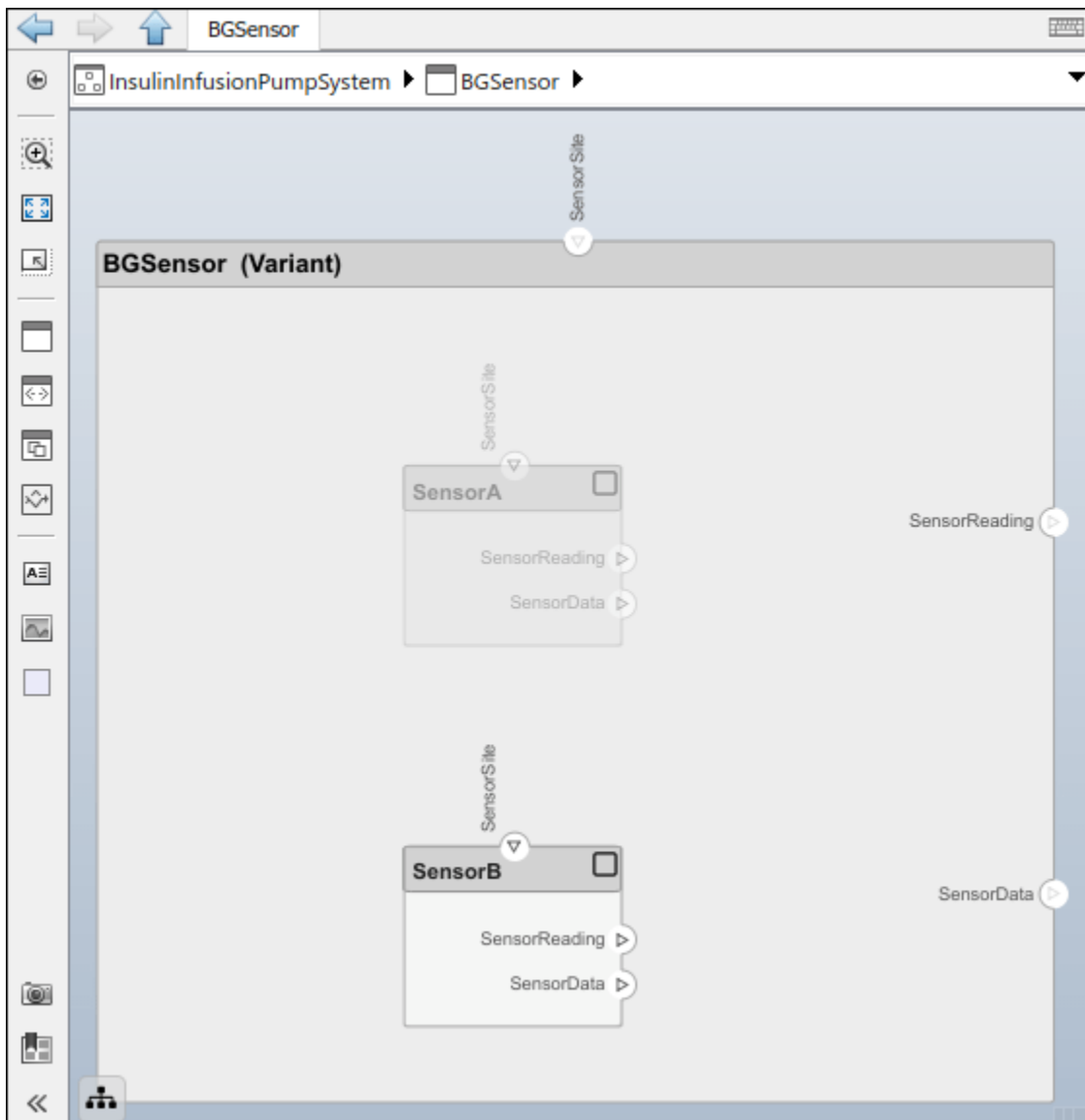
	Property name	Type	Name	Unit	Default
1	UnitCost	double	n/a	\$	0
2	NRE	double	n/a	\$	0
3	Size	double	n/a	g	250
4	EnergyConsumption	double	n/a	mW	2
5	ComplianceScore	double	n/a		0
6	Vendor	string	n/a	n/a	
7	MTBF	double	n/a	hour	0
8	Precision	double	n/a		0
9	Tolerance	double	n/a		0

Show inherited properties (read-only)

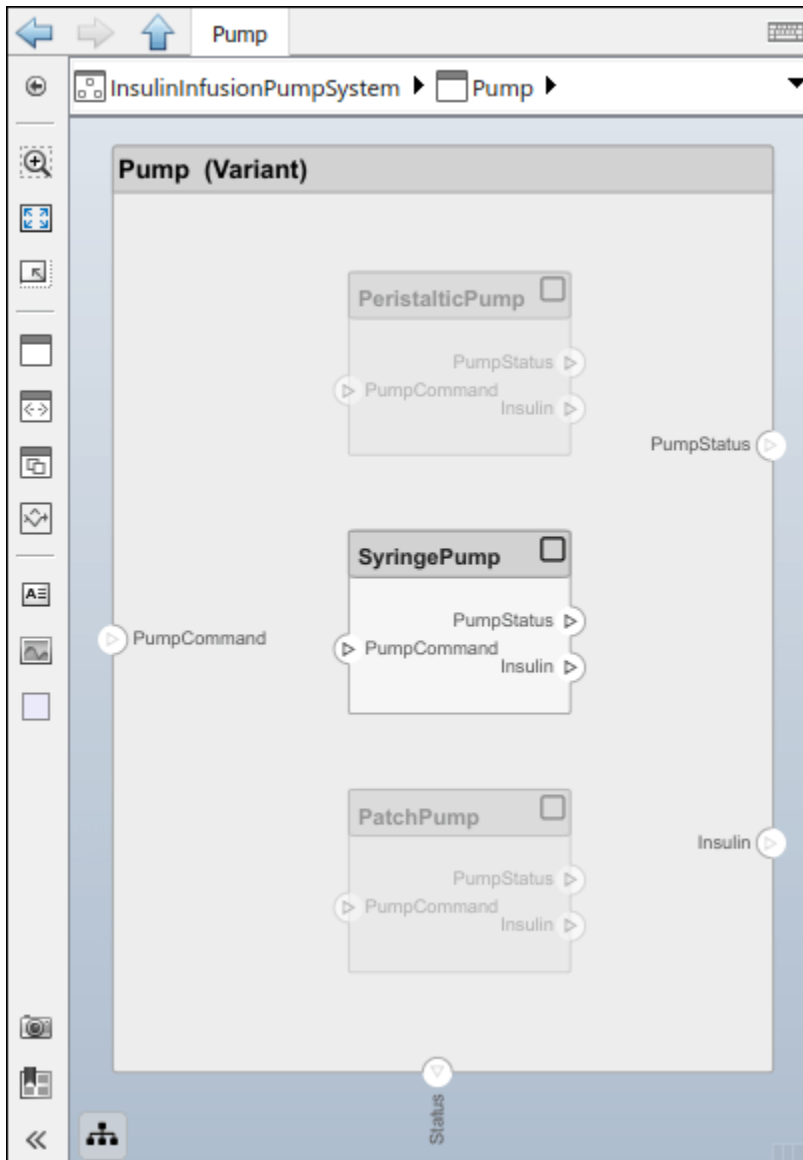
The pump and sensor trade study includes these steps:

- 1 Collect all variant combinations.
- 2 Activate variants one by one to represent all the combinations.
- 3 Iterate over the model to calculate compliance and compute the outcome using the stored and calculated parameters.
- 4 Collect outcomes and weight them using the same units.
- 5 Provide the optimized option.

A Variant Component block named **BGSensor** contains two different sensor variants representing example sensors from different manufacturers.

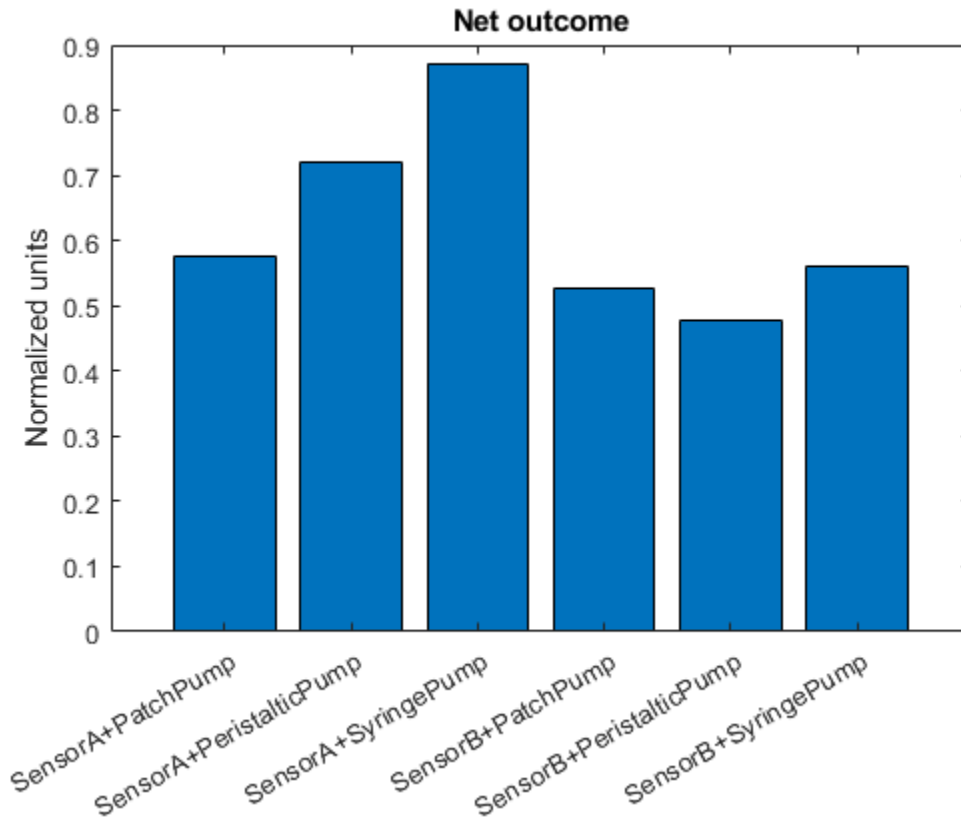


The Variant Component block named **Pump** contains three different pumps in this example called **PeristalticPump**, **SyringePump**, and **PatchPump**.



To programmatically cycle between the different variant choice combinations, calculate compliance, and monitor the outcome to determine the optimal design choice, run `OutcomeAnalysis.m`. For more information on variant analysis, see “Analysis Function Constructs” on page 7-9.

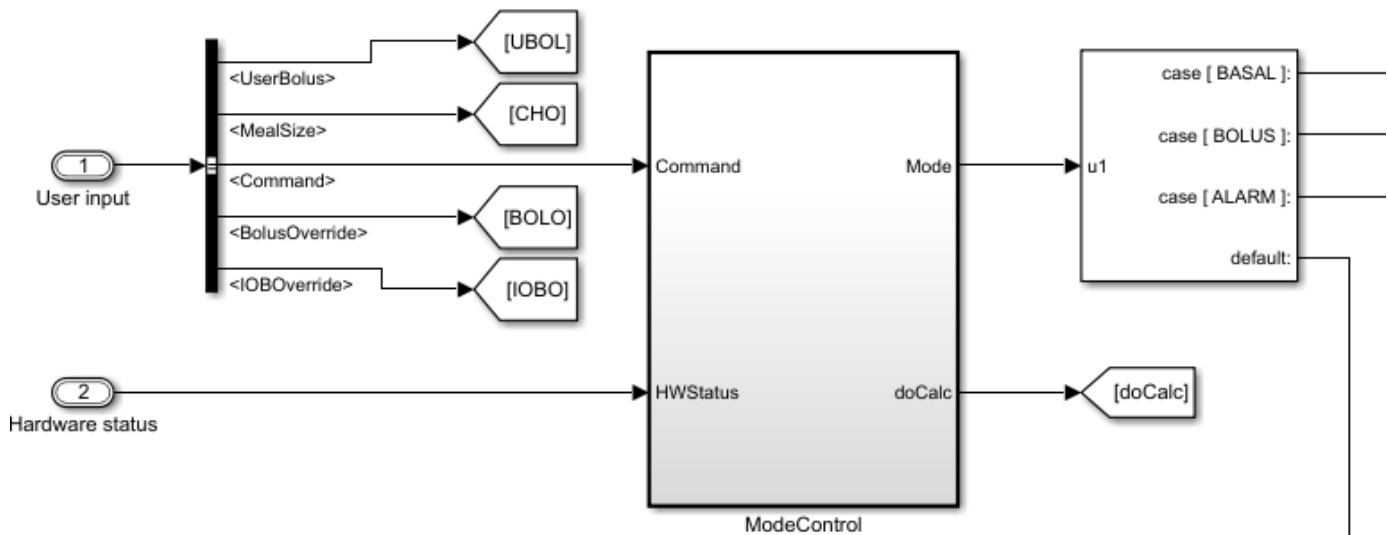
```
run("OutcomeAnalysis.m")
```



The normalized compliance score is at a maximum for the SensorA + SyringePump combination. This design choice is optimal for the insulin pump.

Controller Implementation Model

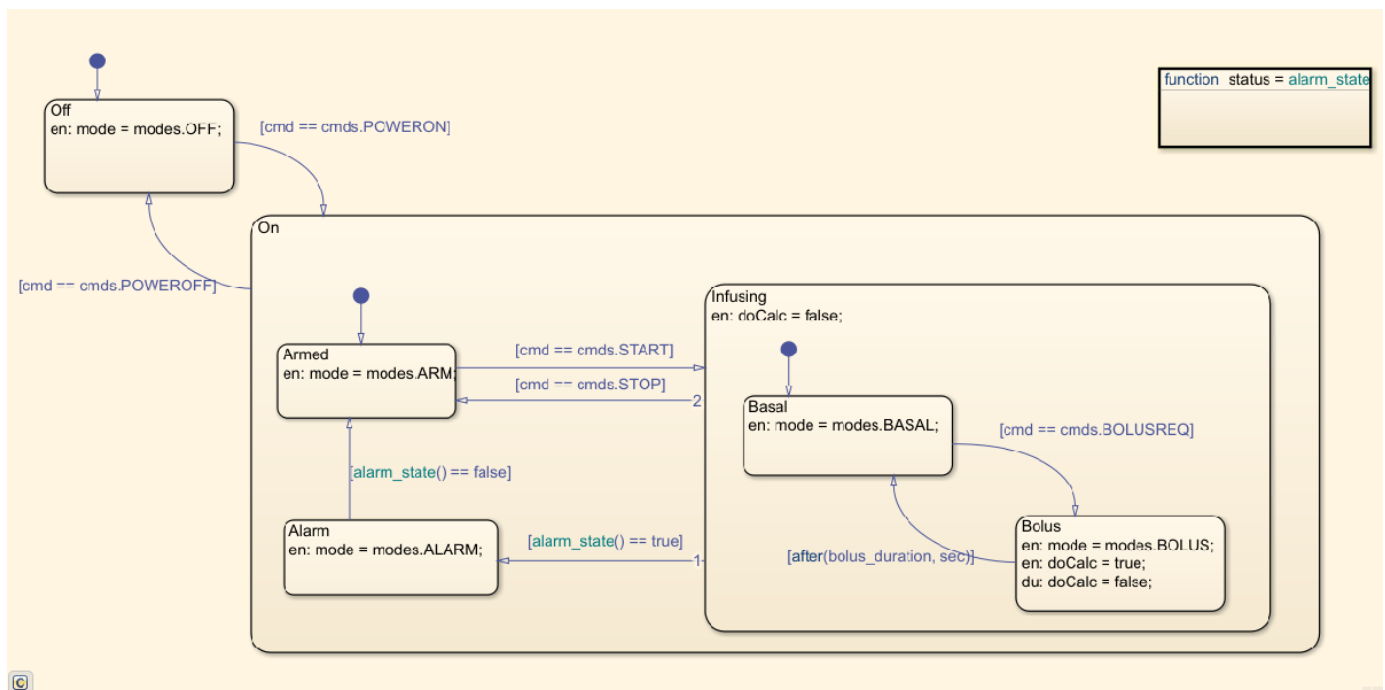
Implement the insulin infusion pump controller in Simulink®. The input ports in this implementation include `User input`, with user metrics that the insulin pump reads, and `Hardware status`, with information about the insulin pump. The block named `ModeControl` determines in which mode the insulin pump must operate.



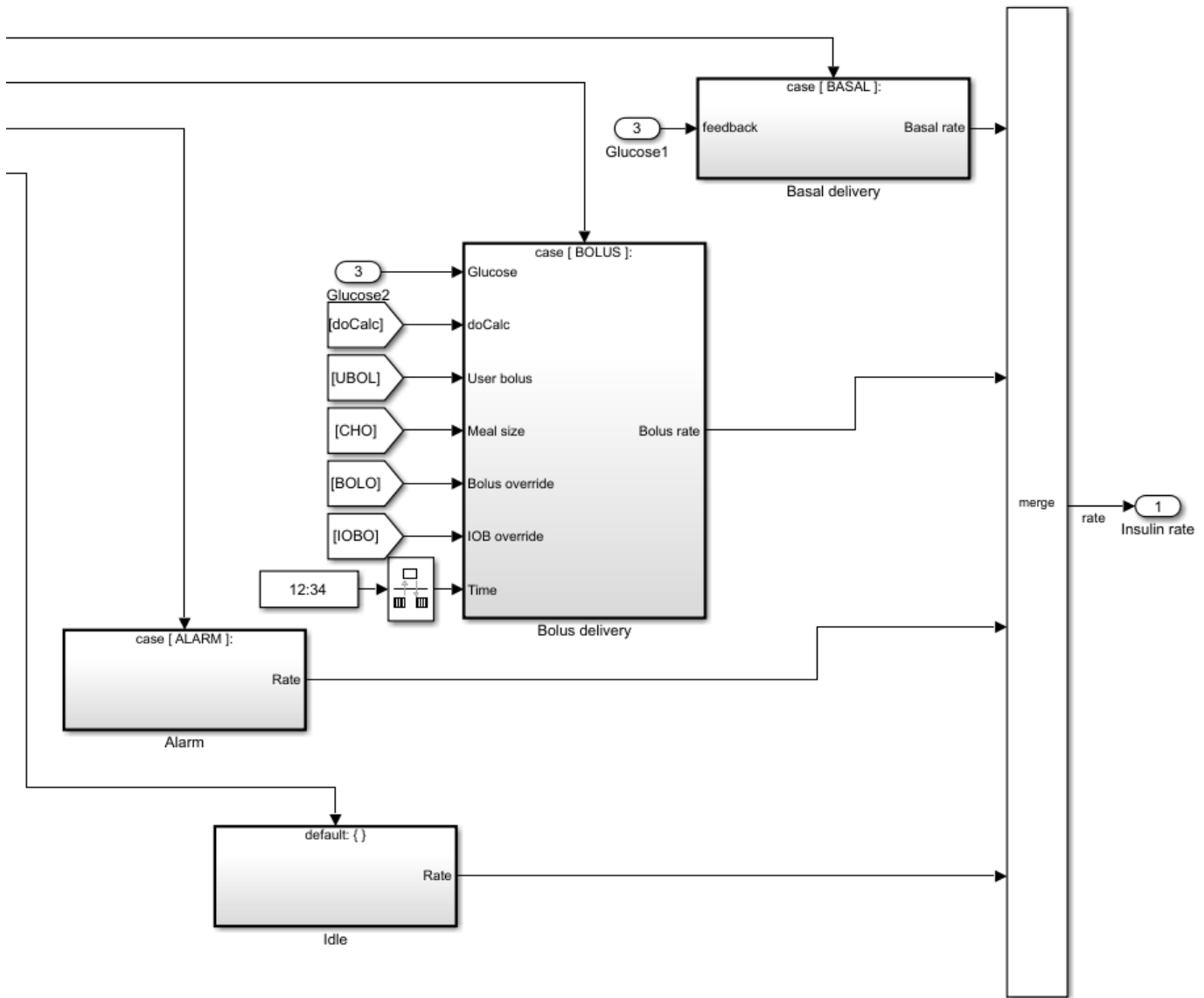
The block named ModeControl contains a Stateflow chart with details on how to select the mode.

The three modes include:

- Alarm mode, where the system is suspended, repaired, and restarted once clear
- Bolus delivery mode to deliver insulin quickly with food intake
- Basal delivery mode to deliver insulin over a longer period of time to keep glucose levels steady throughout the day



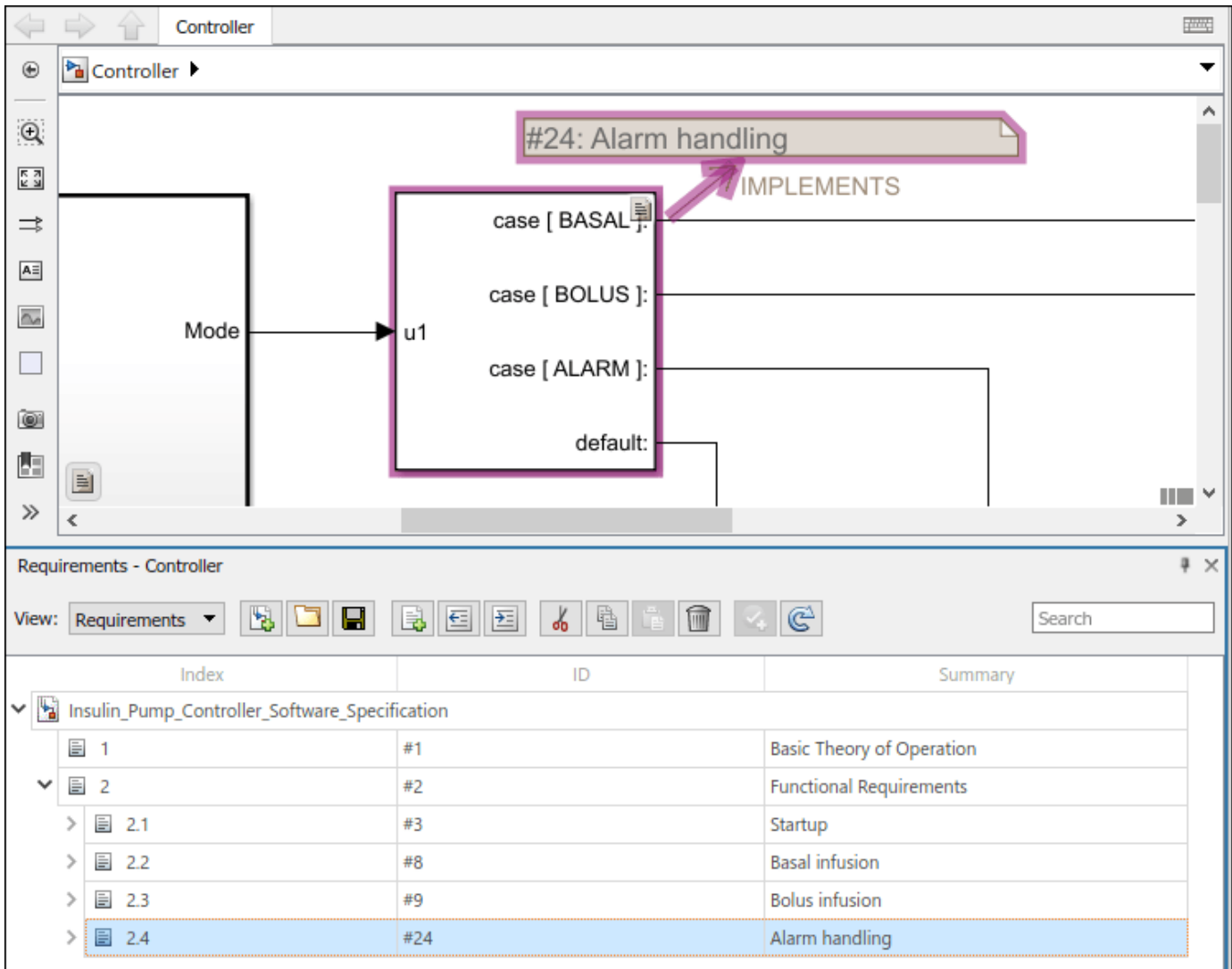
After the mode is selected, this component behavior determines the insulin rate for the output.



Verification and Validation Using Test Manager

You can use model-based design to verify architectural designs and system requirements. The abstract architecture model and the detailed Simulink design model are connected with traceable requirement links.

The Controller implementation model in Simulink demonstrates requirements traceability for the Alarm handling requirement.



Load and view the Test Manager (Simulink Test) using these commands.


```
sltest.testmanager.load("Controller_Tests.mldatx");
sltest.testmanager.view
```

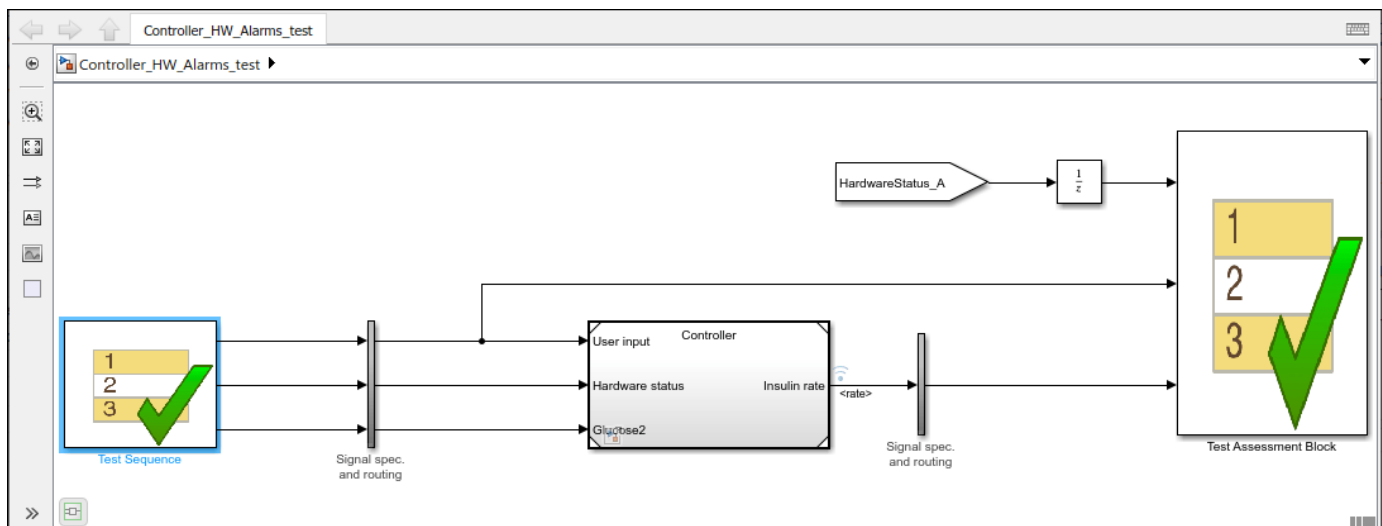
The Alarm_Detection functional test verifies the Alarm handling requirement.

The screenshot shows the Test Browser interface with the following details:

- Test Browser:** Filter tests by name or tags, e.g. tags: t. The tree view shows Controller_Tests > Functional_Tests > Alarm_Detection.
- Property Table:**

PROPERTY	VALUE
Name	Alarm_Detection
Type	Baseline Test
Model	Controller
Harness Name	Controller_HW_A...
Simulation Mode	[Model Settings]
Location	C:\Users\lakhan\...
Enabled	<input checked="" type="checkbox"/>
Hierarchy	Controller_Tests ...
Tags	Type comma or spa
- Alarm_Detection Configuration:**
 - Enabled:
 - Controller_Tests » Functional_Tests » Alarm_Detection
 - Baseline Test
 - Create Test Case from External File
 - TAGS
 - DESCRIPTION
 - REQUIREMENTS*
 - Hardware alarms
 - Alarm action
 - Alarm handling
 - SYSTEM UNDER TEST*
 - Model: Controller
 - TEST HARNESS*
 - Harness: Controller_HW_Alarms_test
 - SIMULATION SETTINGS AND RELEASE OVERRIDES

Click the  icon to the right of the **Harness** box to open the test harness. In this example, the block named **Controller** is isolated for unit testing using a test harness. For more information on creating a test harness, see “Create Test Harnesses and Select Properties” (Simulink Test).



Double-click the Test Sequence block to view the steps in the test sequence. The steps define a scenario to verify the functioning of the alarm system.

Step	Transition	Next Step	Description
Run %% Initialize data outputs. UserInput.Command = cmds.POWEROFF; UserInput.MealSize = 0; UserInput.UserBolus = 0; UserInput.BolusOverride = false; UserInput.IOBOverride = false; HardwareStatus.BatteryFault = false; HardwareStatus.LowReservoir = false; HardwareStatus.LineBlockage = false; Glucose2 = 0;	1. true	startup	
startup UserInput.Command = cmds.POWERON;	1. after(10,sec)	Infuse	
Infuse UserInput.Command = cmds.START Glucose2 = 10;	1. after(10,sec)	BatteryFault	
BatteryFault HardwareStatus.BatteryFault = true;	1. after(5,sec)	clear1	
clear1 HardwareStatus.BatteryFault = false;	1. true	restart1	
restart1 UserInput.Command = cmds.START;	1. after(5,sec)	OutOfInsulin	
OutOfInsulin HardwareStatus.LowReservoir = true;	1. after(10,sec)	clear2	
clear2 HardwareStatus.LowReservoir = false;	1. true	restart2	
restart2 UserInput.Command = cmds.START;	1. after(5,sec)	Occlusion	
Occlusion HardwareStatus.LineBlockage = true;			

To run this test, go back into the Test Manager (Simulink Test).

```
sltest.testmanager.view
```

Right-click the test `Alarm_Detection` in the Test Browser and select **Run**. In the Results and Artifacts section, view your test results. A passing test indicates that the system requirement `ALarm handling` is verified by the conditions defined in the Test Assessment Block:

- Whether the alarm disables insulin delivery when there is low battery, occlusion (line blockage), or low medication (insulin)
- Whether the system restarts after the issue has passed

See Also

`systemcomposer.profile.editor` | `slreq.editor` | `sltest.testmanager.view`

More About

- “Manage Requirements” on page 2-8
- “Compose Architecture Visually” on page 1-2
- “Analysis Function Constructs” on page 7-9
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22
- “Calculate Endurance Using Quadcopter Architectural Design” on page 7-16

Software Architectures

- “Author Software Architectures” on page 8-2
- “Simulate and Deploy Software Architectures” on page 8-8
- “Modeling the Software Architecture of a Throttle Position Control System” on page 8-14
- “Class Diagram View of Software Architectures” on page 8-20
- “Author and Extend Functions for Software Architectures” on page 8-24
- “Merge Message Lines Using Adapter Block” on page 8-32
- “Authoring Functions for Software Components of an Adaptive Cruise Control” on page 8-34
- “Author Service Interfaces for Client-Server Communication” on page 8-41
- “Service-Oriented Sensor Modeling” on page 8-44

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 New Software Architecture Model

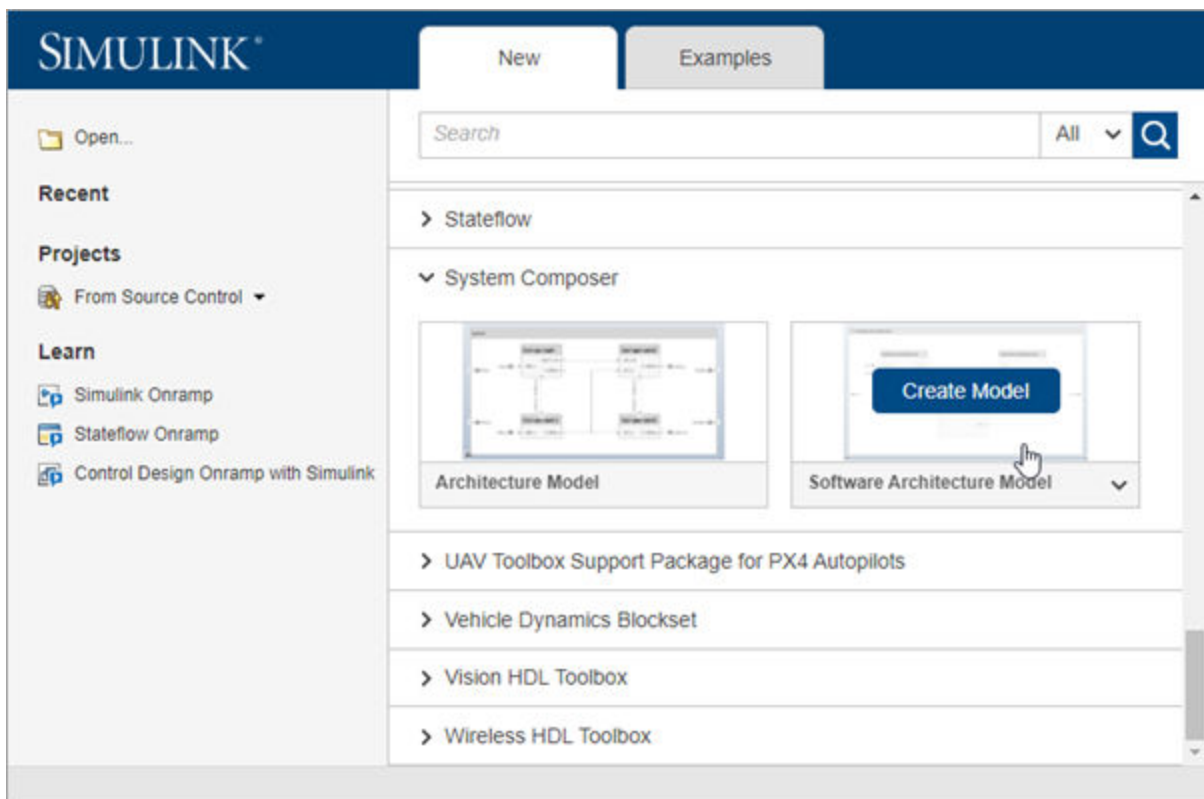
The workflow for authoring software architecture models is similar to authoring system architectures. Start with a blank software architecture template to model.



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

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

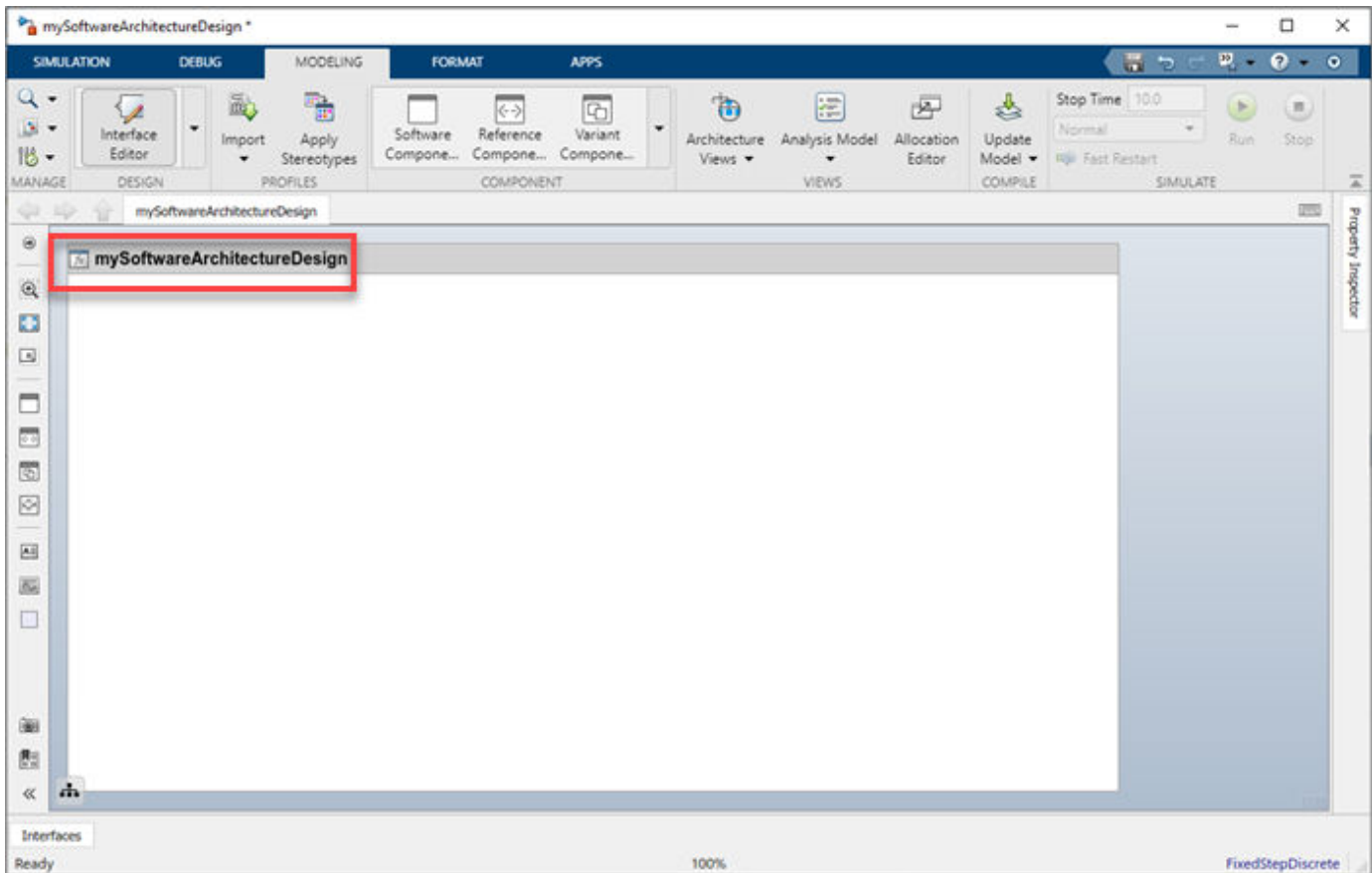
where mySoftwareArchitectureDesign is the name of the new model.

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.

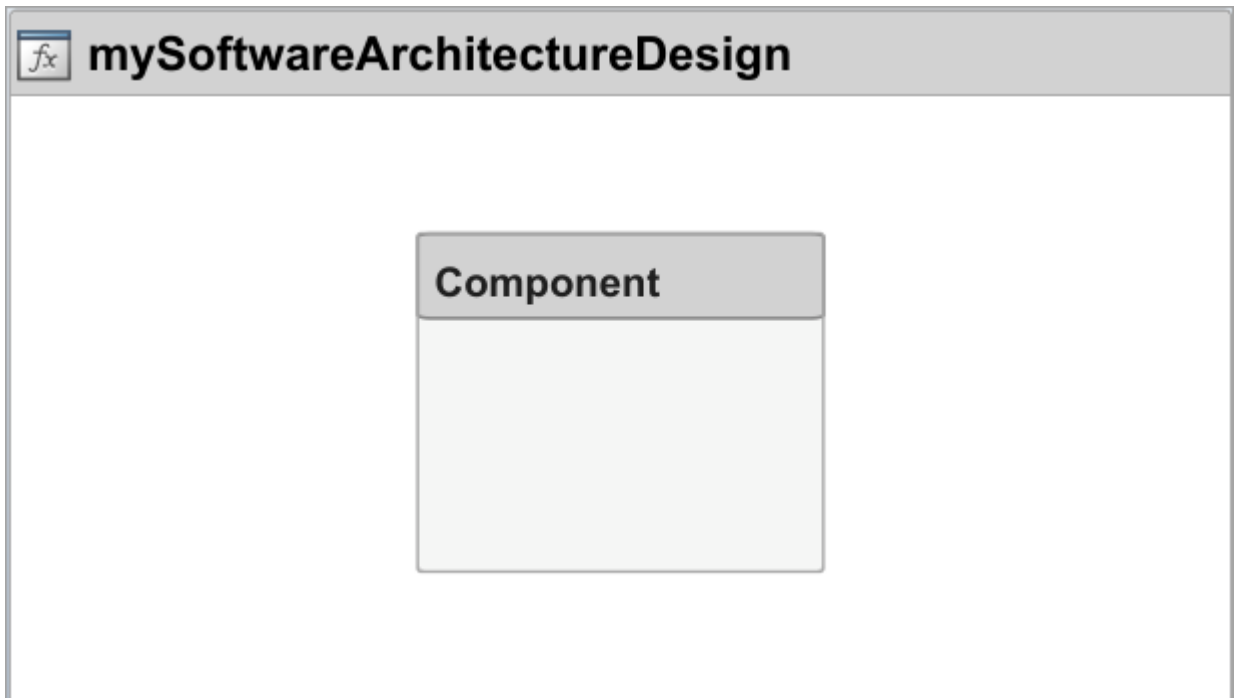


When you model software architectures, you can:

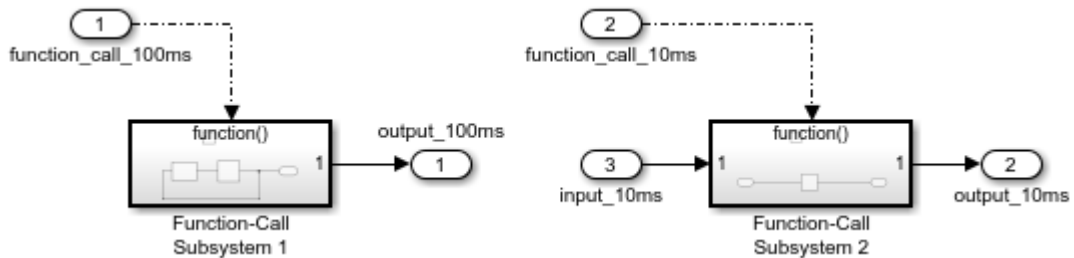
- Use model building and visualization tools provided by System Composer such as components, connections, and ports. For more information, see “Compose Architecture Visually” on page 1-2.
- Define interfaces. For more information, see “Create Interfaces” on page 3-5.
- Create custom views. For more information, see “Create Architecture Views Interactively” on page 9-5.
- Use tools to write analysis and create allocations. For more information, see “Analyze Architecture” on page 7-2 and “Create and Manage Allocations” on page 6-2.

Build a Simple Software Architecture Model

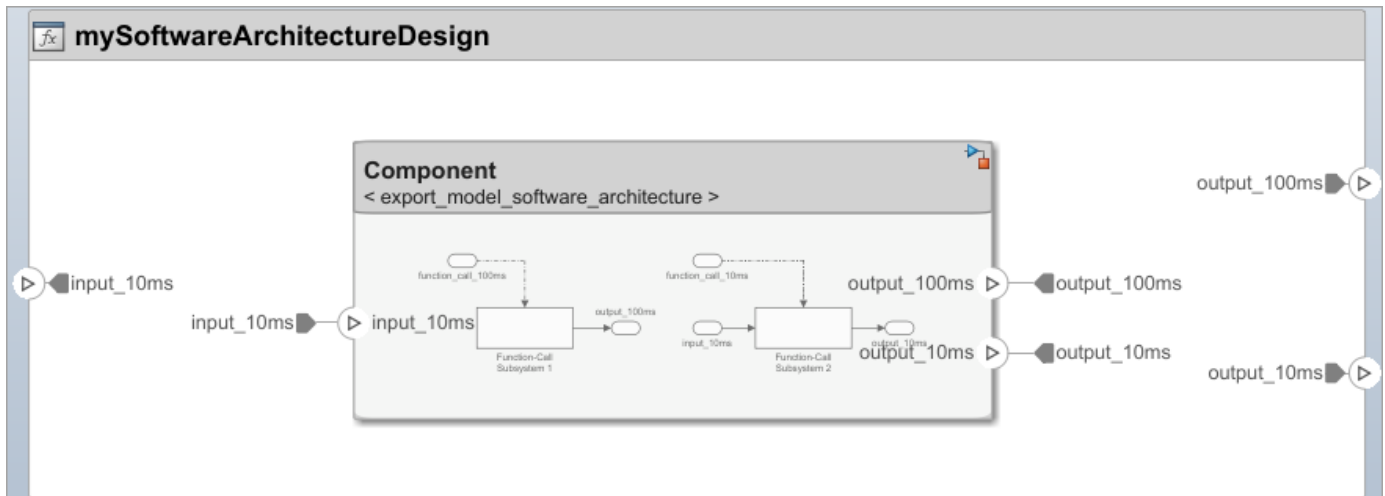
- 1 Drag an empty component to the mySoftwareArchitectureDesign model.



- 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 8-8.

Import and Export Software Architectures

You can import a software architecture model using the `systemcomposer.importModel` function.

```
archModel = systemcomposer.importModel(modelName,importStruct)
```

If the domain field of `importStruct` is "Software", the `importModel` function creates a new software architecture based on the structure of the MATLAB tables.

To export a System Composer software architecture model, use the `systemcomposer.exportModel` function.

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

The `exportModel` function returns a structure containing MATLAB tables that contains components, ports, connections, portInterfaces, requirementLinks, and a domain field with value 'Software' to indicate that the exported architecture is a software architecture.

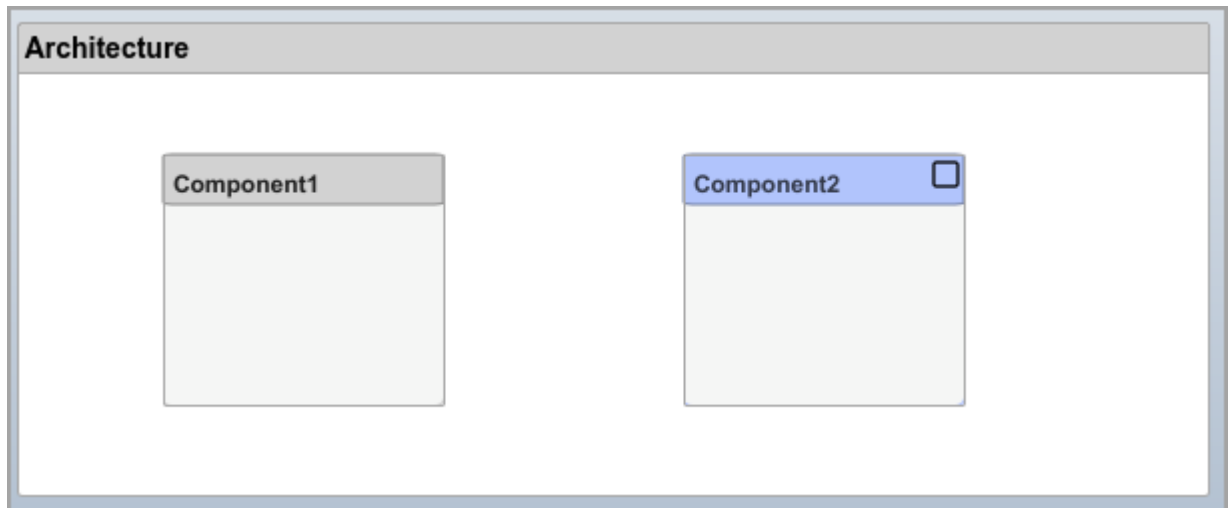
For more information on importing and exporting software architectures with functions, see “Import and Export Functions of Software Architectures” on page 8-30.

Create Software Architecture from Architecture Model Component

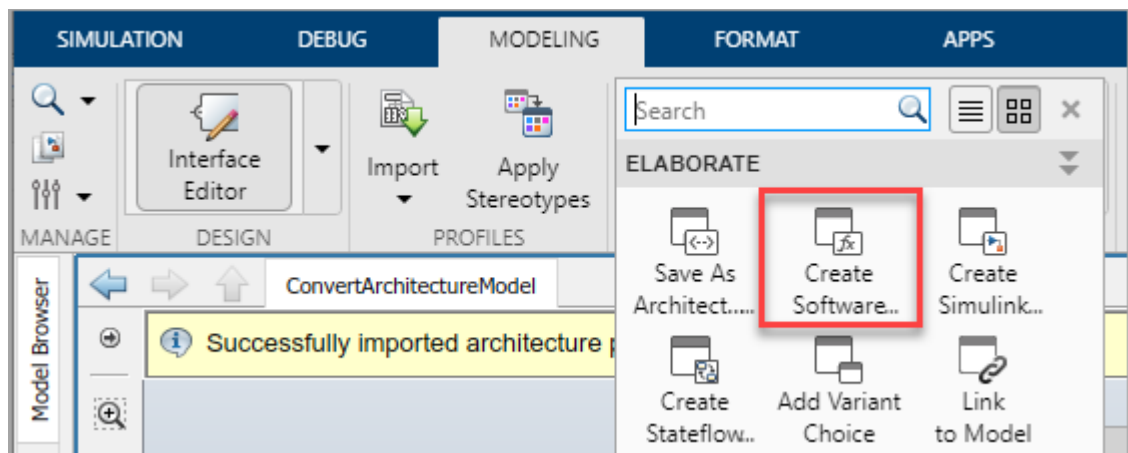
You can also create a software architecture model from an existing component in a System Composer architecture model.

To create a software architecture model from a component:

- 1 Select an existing component from your architecture model. In this example, we select Component2.



- 2 To create a software architecture model from Component2, you can use any of these three methods:
 - a Right-click the component and select **Create Software Architecture Model**.
 - b Select the component and, on the toolbar, click **Create Software Architecture Model**.



- c To create a software architecture programmatically, use the `createArchitectureModel` function.
- 3 Observe the software architecture model icon in the upper left corner. The new software architecture contains all elements from the component, including previously applied stereotypes.



The following elements are not supported if you create a software architecture from an existing component:

- A reference component that references a system architecture.
- A component with Stateflow chart behavior.
- Adapter blocks with applied interface conversions. “Interface Adapter” on page 3-16 conversions are removed when you create a software architecture from an existing component.

See Also

`systemcomposer.createModel` | `createArchitectureModel` | `createSimulinkBehavior`

More About

- “Compose Architecture Visually” on page 1-2
- “Create an Export-Function Model”
- “Class Diagram View of Software Architectures” on page 8-20
- “Modeling the Software Architecture of a Throttle Position Control System” on page 8-14
- “Simulate and Deploy Software Architectures” on page 8-8

Simulate and Deploy Software Architectures

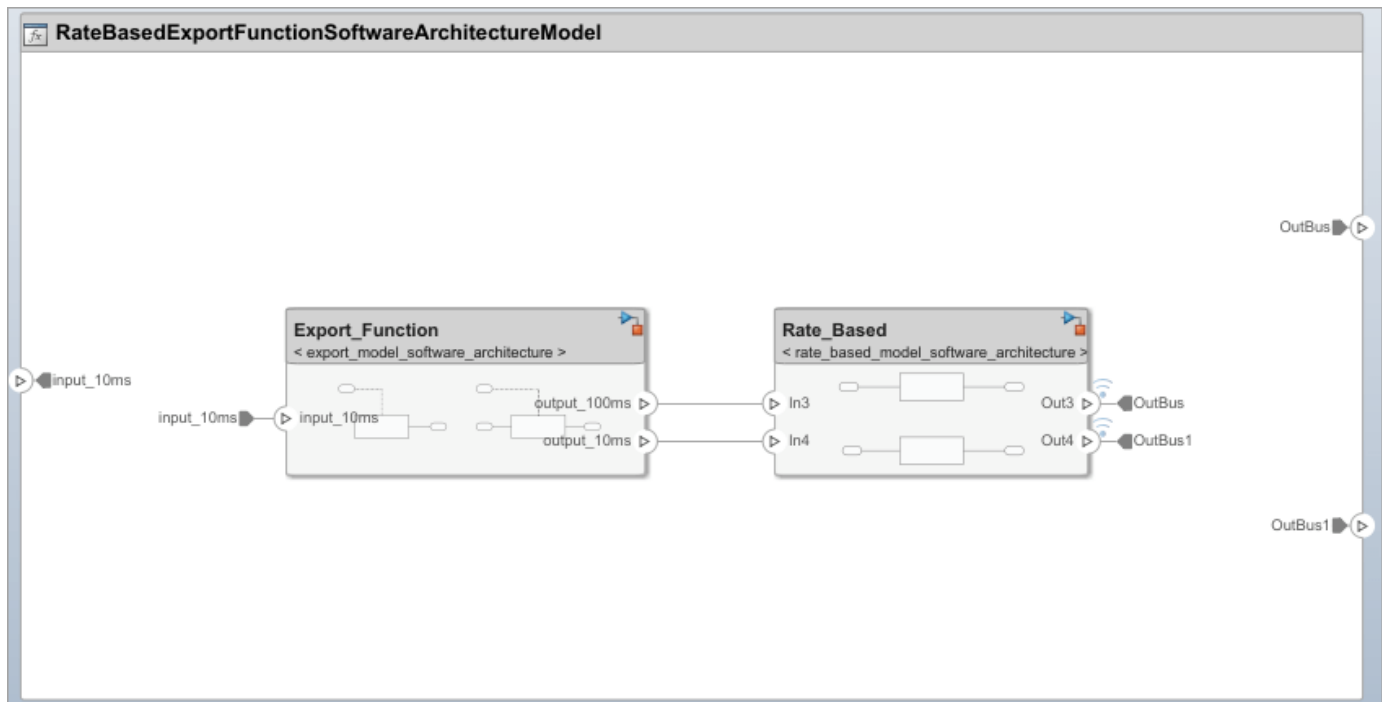
This example shows how to build a multi-component software architecture model with a rate-based and export-function components, how to simulate your design at the architecture level, and how to generate code.

Open the Software Architecture Model

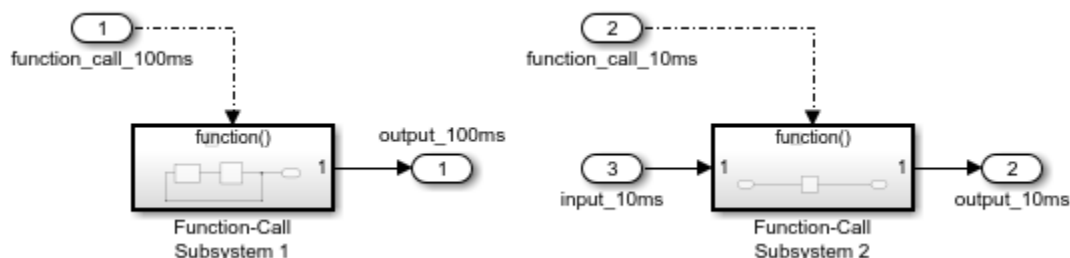
After opening the example, open the model below. This software architecture model has two software components: `Export_Function` and `Rate_Based`.

```
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`.

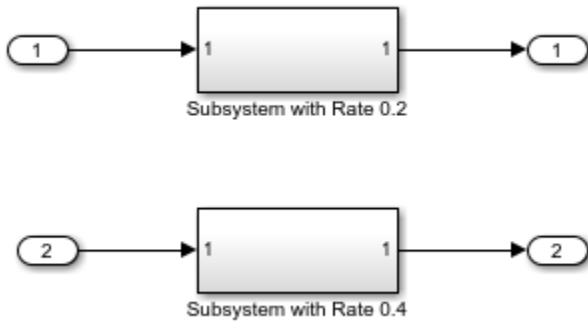


In this Simulink behavior, two functions are modeled using Function-Call Subsystem blocks. The input blocks are connected to the function-call input ports and generate periodic function-call events with sample times 10ms and 100ms. To learn how to model this behavior, see “Create an Export-Function Model”.



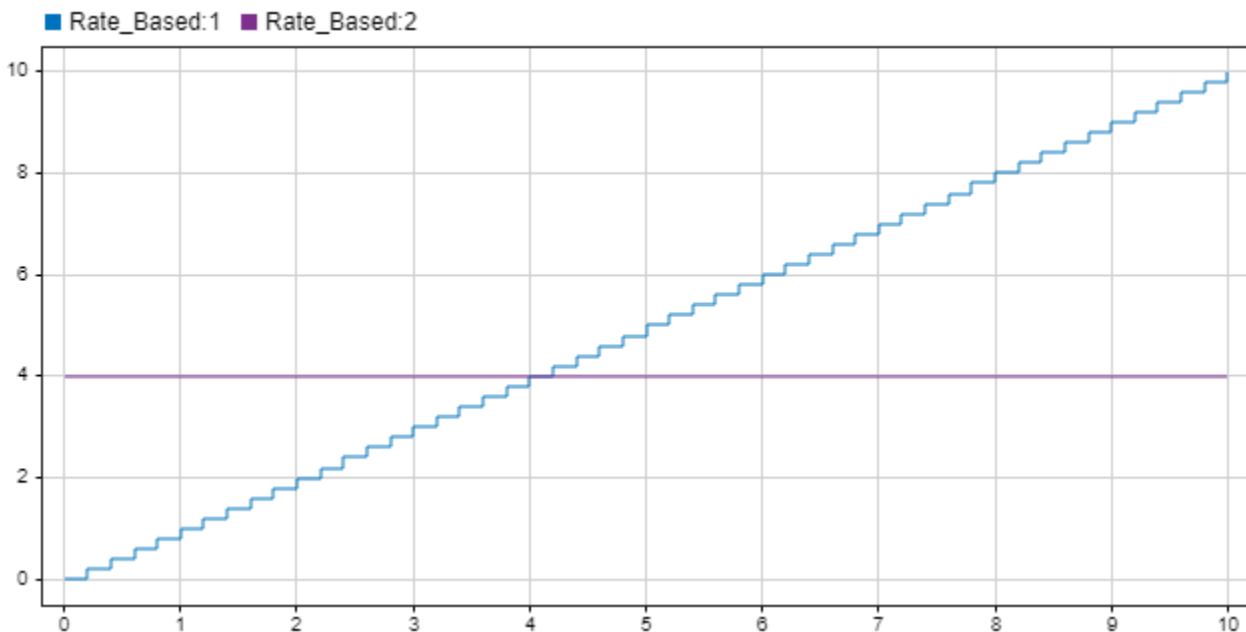
If the inport blocks that are connected to the function-call input ports with sample time specified as -1, meaning the functions are aperiodic, use a Simulink test model with explicit scheduling blocks such as a Stateflow chart to simulate. For more information see Test Software Architecture on page 8-0 .

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

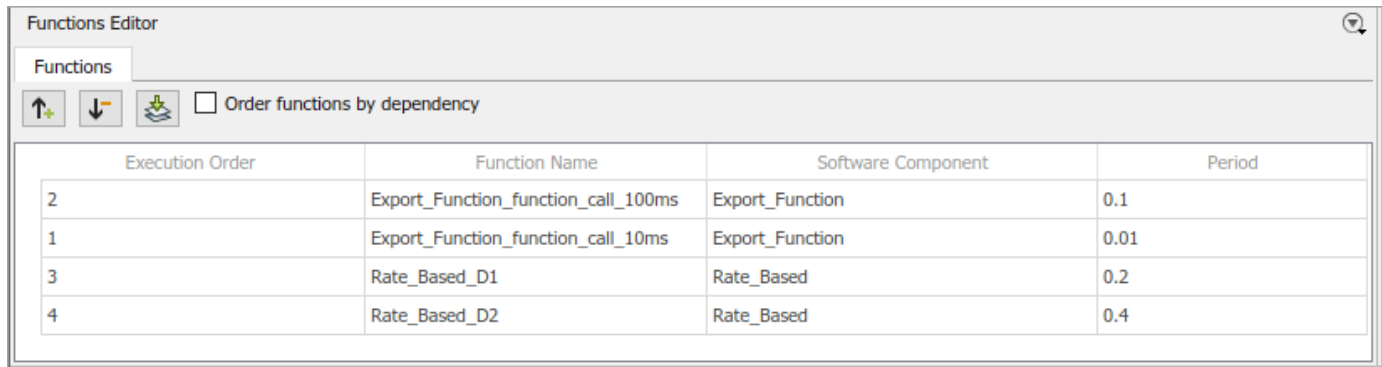
Simulate the model. Observe that the Simulation Data Inspector displays the output from the Rate-Based component.



Visualize and Edit Component Functions Using Functions Editor

Use the Functions Editor to edit simulation execution order of the functions in your software architecture. You can also edit the sample time of the functions with inherited sample time (-1).

The Functions Editor is visible only when you model software architectures. To open the Functions Editor, in the toolstrip on the **Modeling** tab, select **Functions Editor**.



The screenshot shows the 'Functions Editor' window. At the top, there is a 'Functions' tab and a toolbar with an up arrow, a down arrow, a refresh icon, and a checkbox labeled 'Order functions by dependency'. Below the toolbar is a table with the following data:

Execution Order	Function Name	Software Component	Period
2	Export_Function_function_call_100ms	Export_Function	0.1
1	Export_Function_function_call_10ms	Export_Function	0.01
3	Rate_Based_D1	Rate_Based	0.2
4	Rate_Based_D2	Rate_Based	0.4

To edit the functions in your software architecture:

- 1 Open the Functions Editor. When you open the Functions Editor, the model will automatically update, and the table will display the functions populated from your model.
- 2 If there are changes in the software architecture model, the **Update Model** button becomes yellow to signal that an update is required to refresh your functions table.
- 3 To arrange the execution order of the functions, use the up and down arrows or drag and drop functions to sort them.
- 4 To edit sample times of the functions, specify their period in the table.

To order functions based on their data dependencies, select the **Order functions by dependency** check box. To enable sorting of functions based on dependencies, you can set this parameter.

```
set_param('RateBasedExportFunctionSoftwareArchitectureModel', 'OrderFunctionsByDependency', 'on')
```

The default value for the parameter is off.

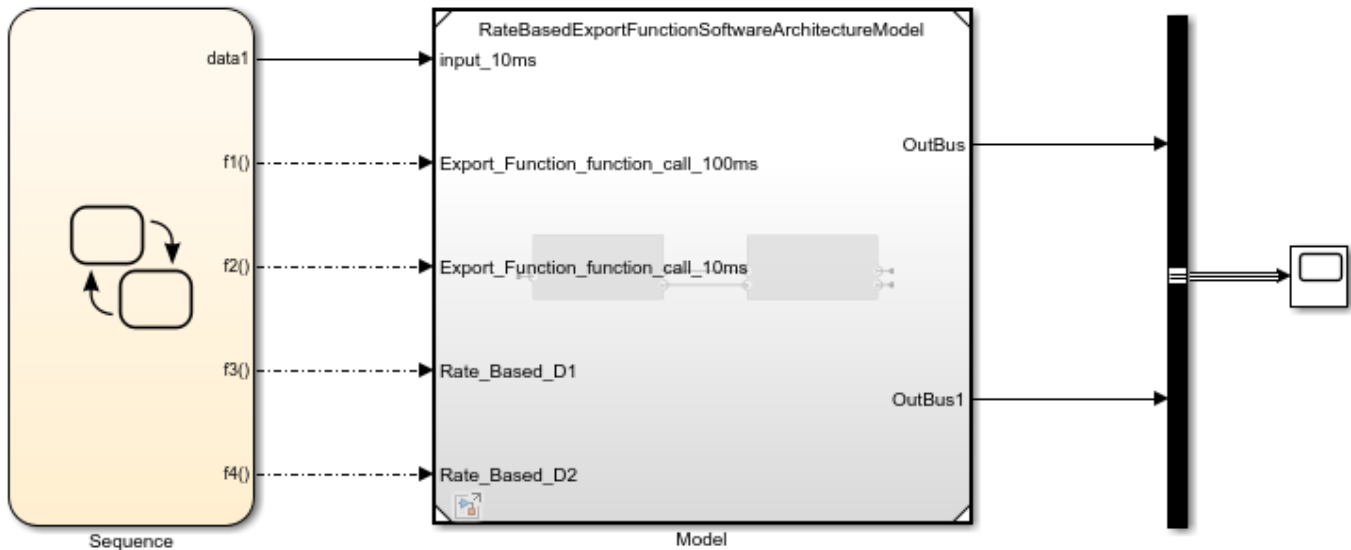
Alternatively, you can use the `systemcomposer.arch.Function` object to get the functions programmatically.

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.



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:
     * Output: '<Root>/OutBus'
     * Output: '<Root>/OutBus1'
     */
    rate_based_model_software_j
        (&RateBasedExportFunctionSoftwareArchitectureModel_B.Export_Function_01,
         &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:
     * Output: '<Root>/OutBus'
     * Output: '<Root>/OutBus1'
     */
    rate_based_model_software_ja
        (&RateBasedExportFunctionSoftwareArchitectureModel_B.Export_Function_02,
         &RateBasedExportFunctionSoftwareArchitectureModel_Y.OutBus1);

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

```

See Also

[systemcomposer.createModel](#) | [createArchitectureModel](#) | [createSimulinkBehavior](#) | [increaseExecutionOrder](#) | [decreaseExecutionOrder](#)

More About

- “Author Software Architectures” on page 8-2
- “Compose Architecture Visually” on page 1-2
- “Create an Export-Function Model”
- “Create A Rate-Based Model”
- “Class Diagram View of Software Architectures” on page 8-20
- “Modeling the Software Architecture of a Throttle Position Control System” on page 8-14
- “Software Component Modeling”

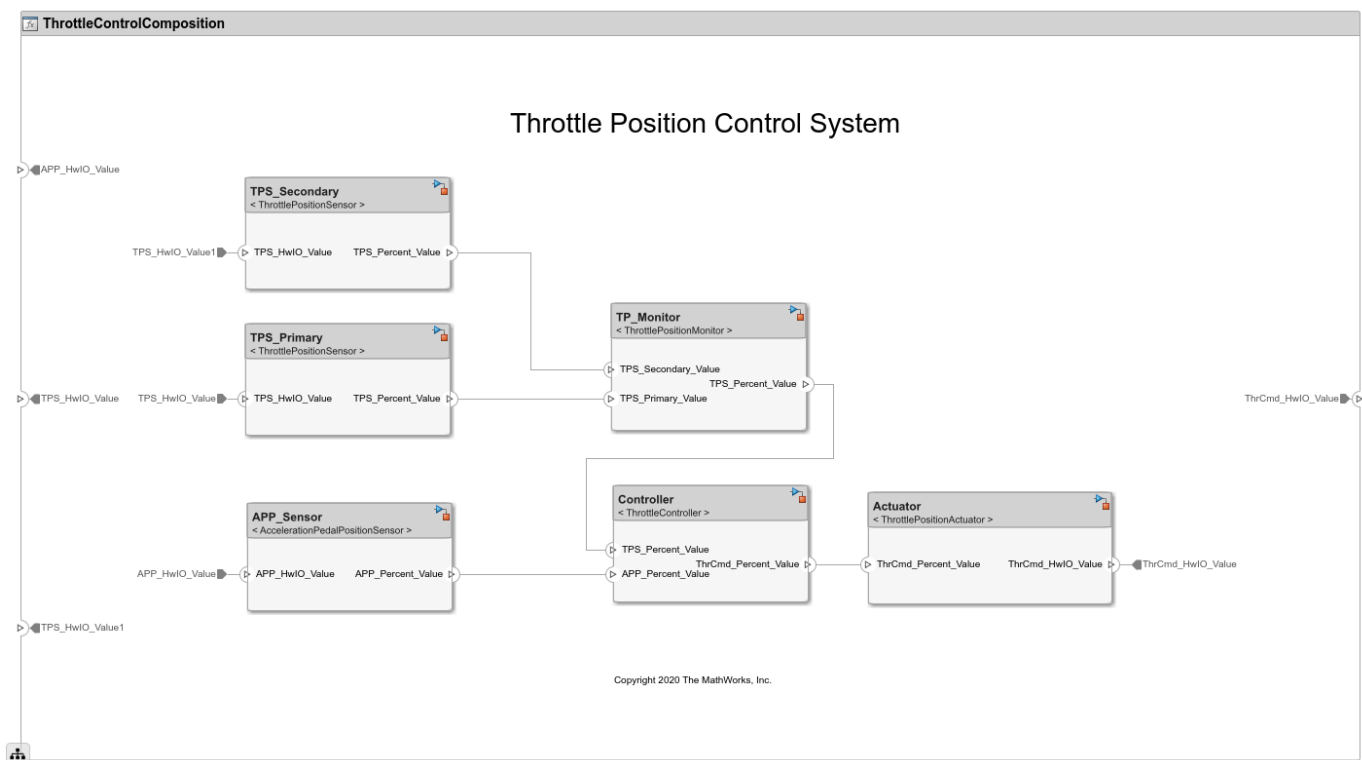
Modeling the Software Architecture of a Throttle Position Control System

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. A controller component uses these signals to determine the new throttle position as a percent value. An actuator component then converts the percent value to the appropriate value for the hardware.

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



Simulate the Model at the Architecture Level

Simulate the software architecture model.

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

To view the list of functions from the components and edit their properties, such as execution order, use the Functions Editor. To open the Functions Editor, on the **Modeling** tab, in the **Design** section,

click **Functions Editor**. For more information about the Functions Editor, see “Simulate and Deploy Software Architectures” on page 8-8.

Functions			
<input type="checkbox"/> Order functions by dependency			
Execution Order	Function Name	Software Component	Period
1	Actuator_output_5ms	Actuator	-1
2	Controller_run_5ms	Controller	0.005
3	TPS_Primary_read_5ms	TPS_Primary	0.005
4	TPS_Secondary_read_5ms	TPS_Secondary	0.005
5	TP_Monitor_D1	TP_Monitor	0.005
6	APP_Sensor_read_10ms	APP_Sensor	0.01

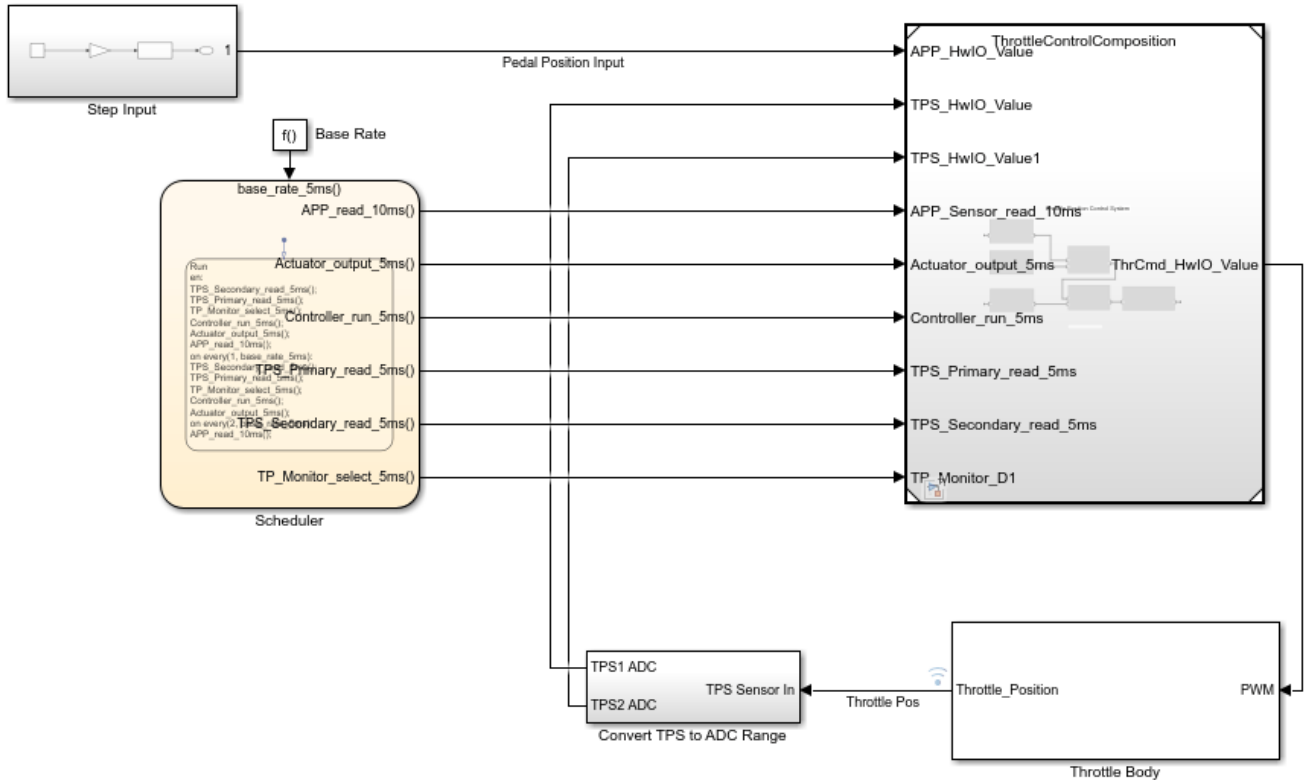
Simulate the Model at the System Level

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.

A Stateflow license is required for this functionality.

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

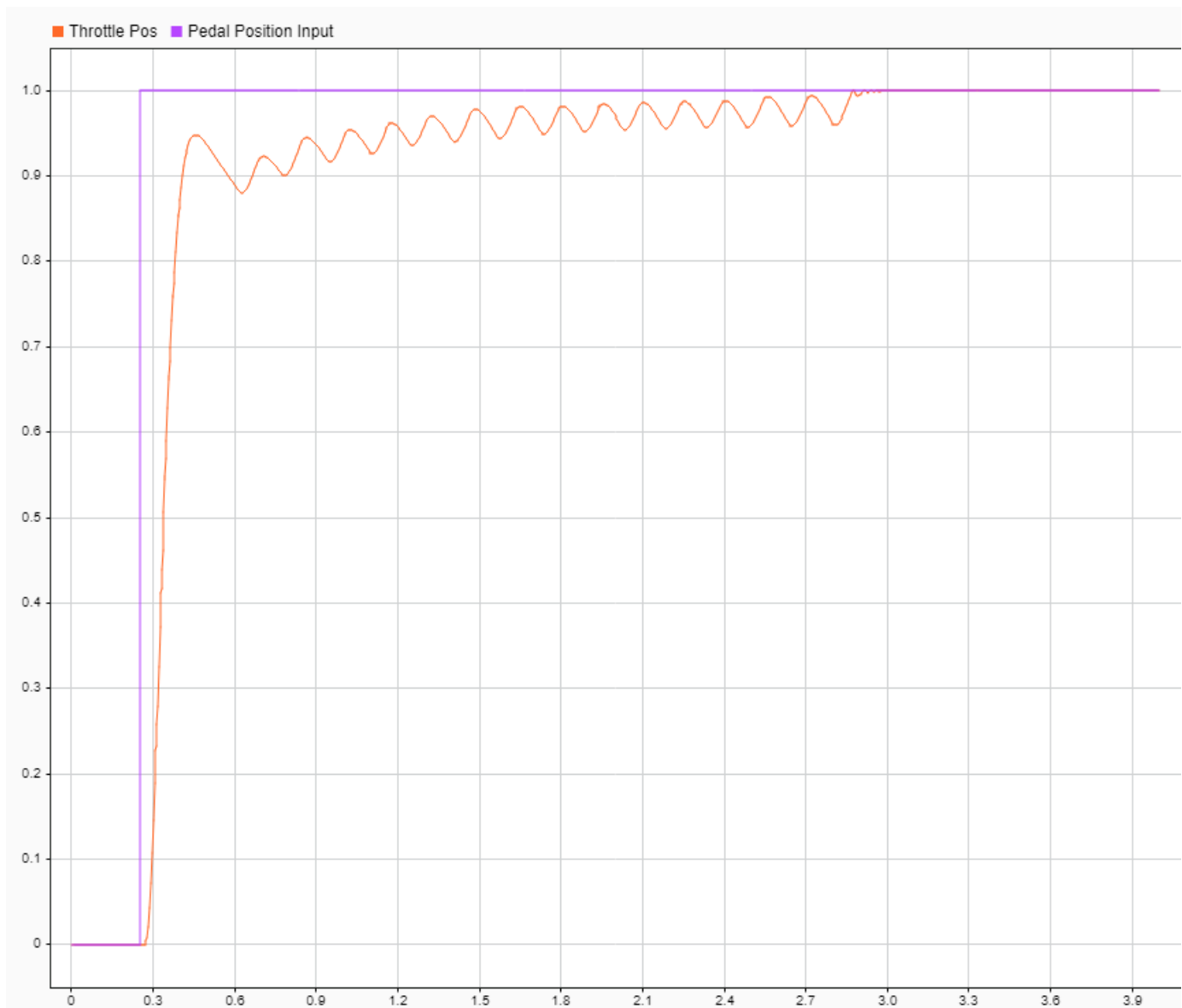
Schedule Functions of a Software Architecture with Stateflow



Copyright 2020-2021 The MathWorks, Inc.

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

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

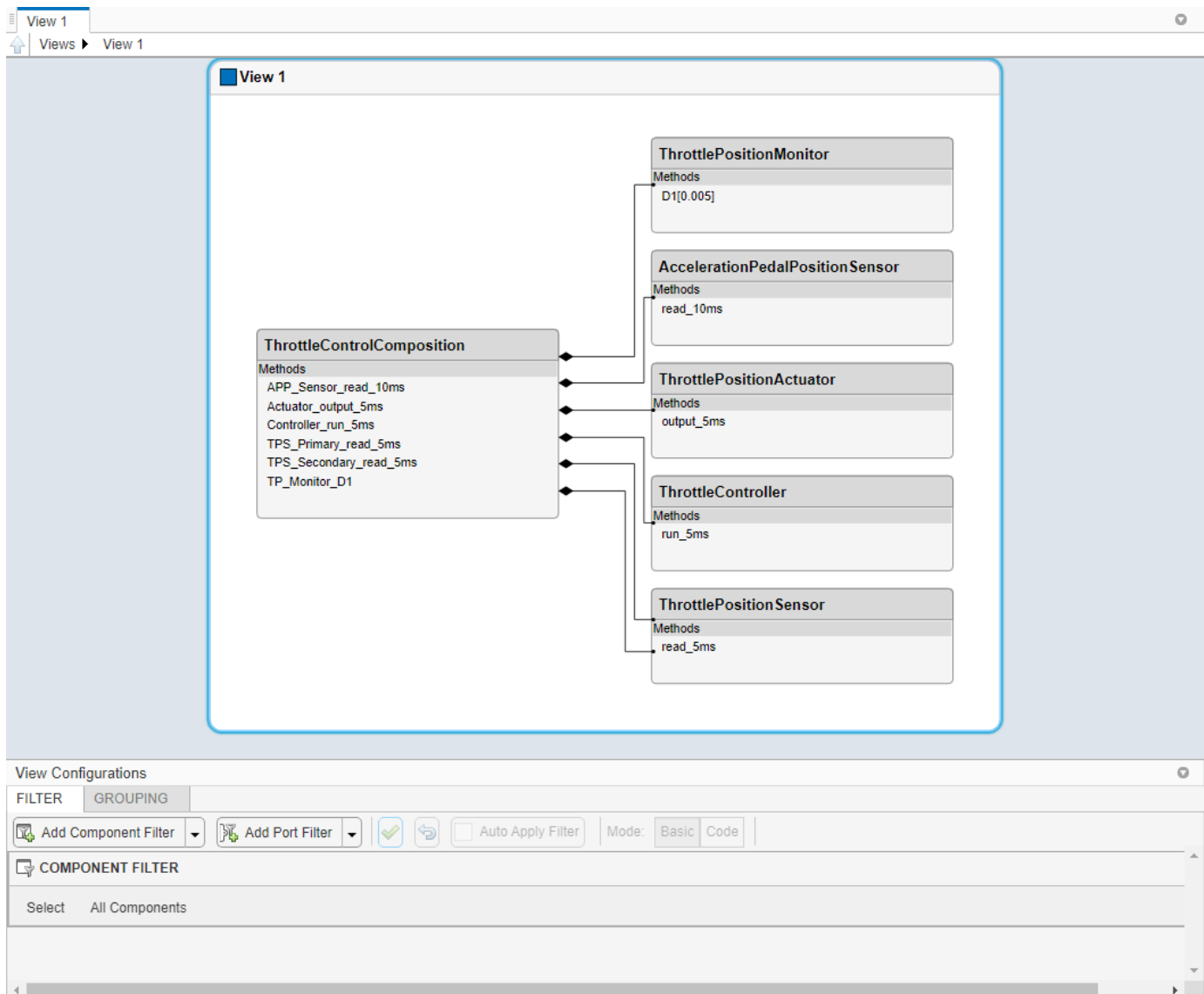


View the Types in the Software Architecture

To view the unique component types in the software architecture, create a class diagram view and add all components. To create a class diagram view, on the **Modeling** tab, in the **Views** section, click **Architecture Views**, then click **New** to create a new class diagram. Select **Class Diagram** from the **Diagram** section in the Views Gallery. From the list, select **Add Component Filter > Select All Components** to add all components in the software architecture to the view.

To populate methods in the class diagram, you must compile the software architecture model. To compile the model, navigate to **Modeling > Update Model**.

For more information, see “Class Diagram View of Software Architectures” on page 8-20.



Code Generation

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”


```
124  /* Model entry point functions */
125  extern void ThrottleControlComposition_initialize(void);
126  extern void ThrottleControlComposition_terminate(void);
127
128  /* Exported entry point function */
129  extern void Actuator_output_5ms(void);
130
131  /* Exported entry point function */
132  extern void Controller_run_5ms(void);
133
134  /* Exported entry point function */
135  extern void TPS_Primary_read_5ms(void);
136
137  /* Exported entry point function */
138  extern void TPS_Secondary_read_5ms(void);
139
140  /* Exported entry point function */
141  extern void TP_Monitor_D1(void);
142
143  /* Exported entry point function */
144  extern void APP_Sensor_read_10ms(void);
145
```

Copyright 2020-2021 The MathWorks, Inc.

See Also

`systemcomposer.createModel` | `createArchitectureModel` | `createSimulinkBehavior` | `increaseExecutionOrder` | `decreaseExecutionOrder`

More About

- “Author Software Architectures” on page 8-2
- “Simulate and Deploy Software Architectures” on page 8-8
- “Class Diagram View of Software Architectures” on page 8-20

Class Diagram View of Software Architectures

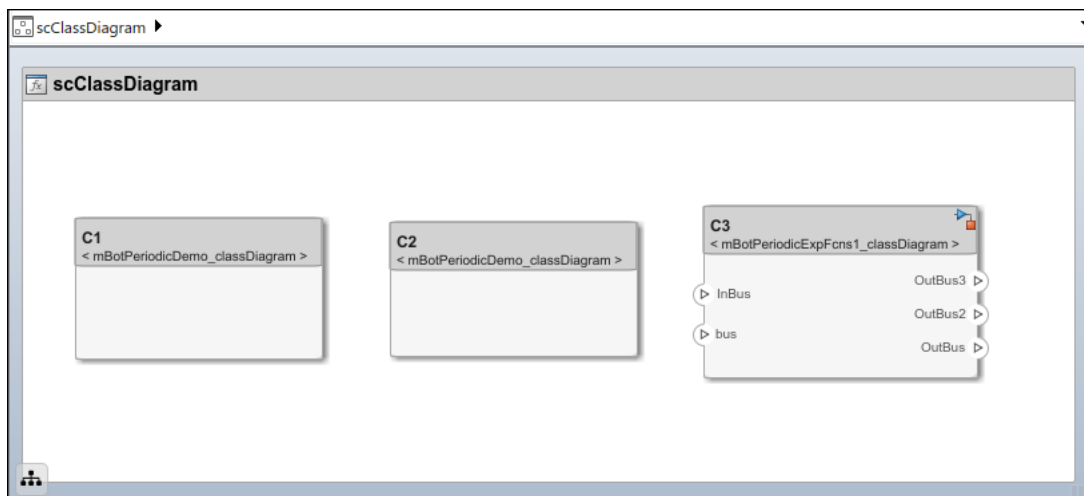
Use class diagrams to display a graphical representation of the structure of a software architecture model. You can also use spotlight views to analyze component dependencies and hierarchy, and you can use component hierarchy views to visualize the component hierarchy as a tree diagram. For more information, see “Create Spotlight Views” on page 9-2 and “Display Component Hierarchy and Architecture Hierarchy Using Views” on page 9-21.

A class diagram is a graphical representation of a static structural model that displays unique architecture types of the software components optionally with software methods and properties.

Class diagrams capture one instance of each referenced model and show relationships between them. Any component diagram view can be optionally represented as a class diagram for a software architecture model.

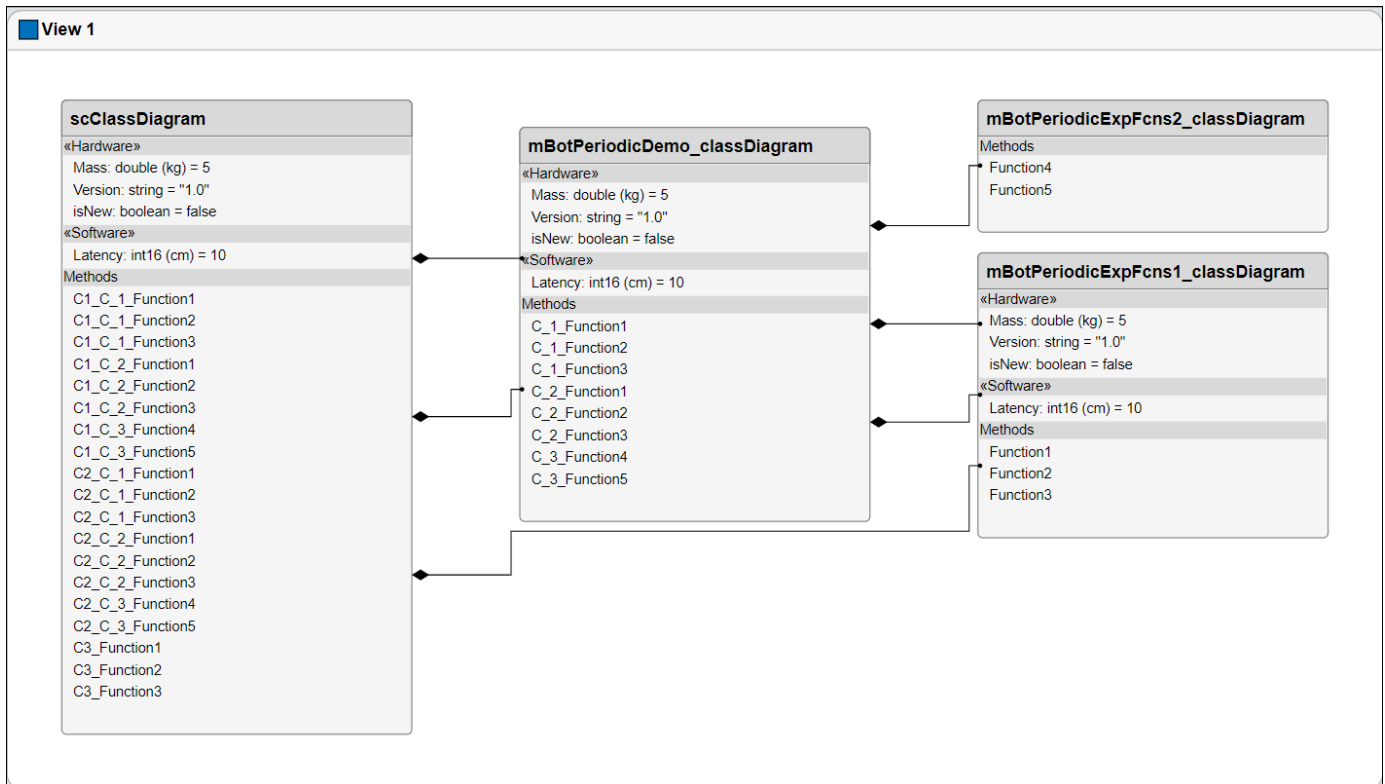
Software Architecture with Class Diagram View

This example uses a software architecture model with functions, stereotypes, and properties to explore class diagrams in the Architecture Views Gallery. Open the model to follow the steps in this tutorial.



Interact with Class Diagram View

- 1 Simulate the model to compile it and populate functions. On the toolbar, click **Run**. Alternatively, update the model to compile it by navigating to **Modeling > Update Model**.
- 2 To open the **Architecture Views Gallery**, navigate to **Modeling > Architecture Views**.
- 3 From the View Browser, select the **View 1** view.
- 4 To open the class diagram view, click **Diagram > Class Diagram**.

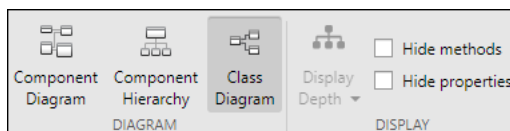


The class diagram consists of:

- A class box for each unique component type, including reference components.
- A class box as the root that corresponds to the root architecture of the top model.
- Composition connections between the types.

If there are multiple instances of the same type of component, for example, multiple components that reference the same model across the model hierarchy, then the type of the component is still represented as one unique box. The component will also relate to its parents and children via multiple composition connections.

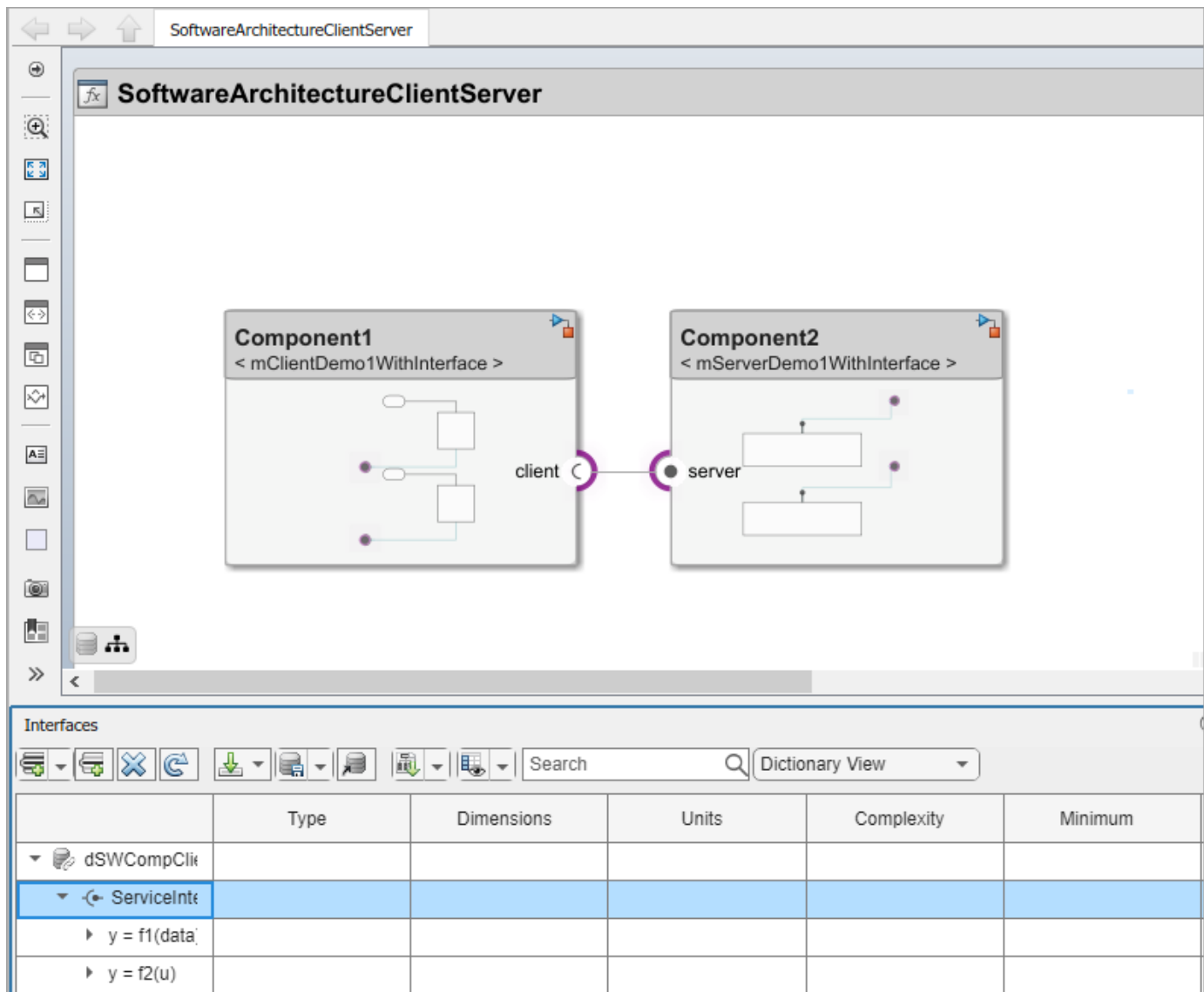
- 5 You can select **Hide methods** to simplify the output by removing software functions from the diagram. Select **Hide properties** to hide information about stereotypes and property values applied to the components.



Client-Server Interfaces in the Class Diagram View

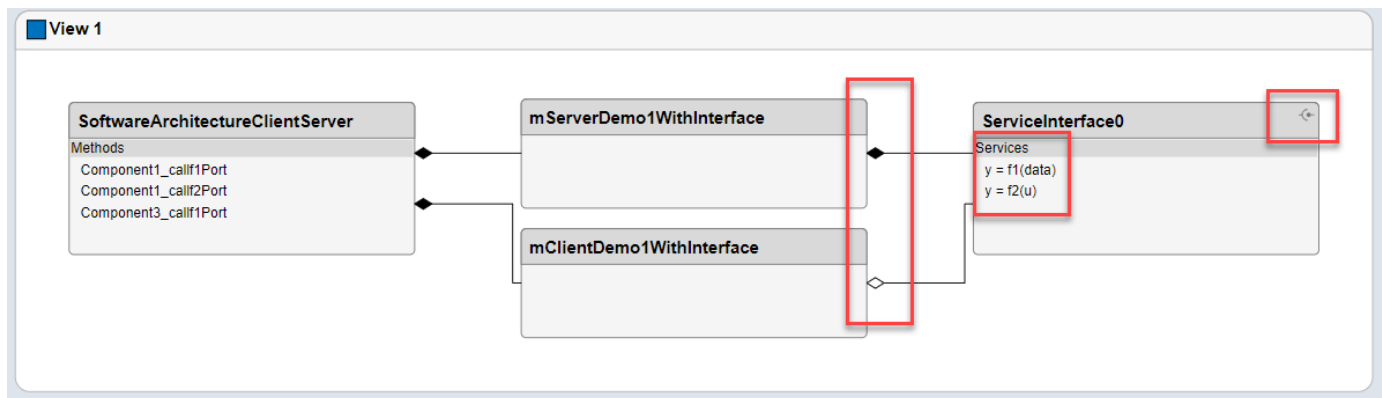
Class diagrams display server (client-server) interfaces. The diagram shows the service interface classes and available services.

In this example, a software architecture has two components that are connected with client and server ports. The **Interface Editor** shows the interface assigned to the client and server ports.



This is the class diagram view of the software architecture.

- The icon on the upper right of the component distinguishes the service interface class.
- The service interface class contains function prototypes as services.
- The diagram displays an aggregation connection for the client port.
- The diagram displays a composition connection for the server port.



See Also

More About

- “Author Software Architectures” on page 8-2
- “Simulate and Deploy Software Architectures” on page 8-8
- “Modeling the Software Architecture of a Throttle Position Control System” on page 8-14
- “Display Component Hierarchy and Architecture Hierarchy Using Views” on page 9-21
- “Author Service Interfaces for Client-Server Communication” on page 8-41

Author and Extend Functions for Software Architectures

For inline components, you can author functions in the architecture level using the **Functions Editor** or by using the `addFunction` function. You can then implement Simulink behaviors for your authored functions.

For reference components, the functions are automatically created from the referenced behavior Simulink models. For information, see “Simulate and Deploy Software Architectures” on page 8-8.

In this topic, we explain the workflow to create functions in the architecture level and describe how to:

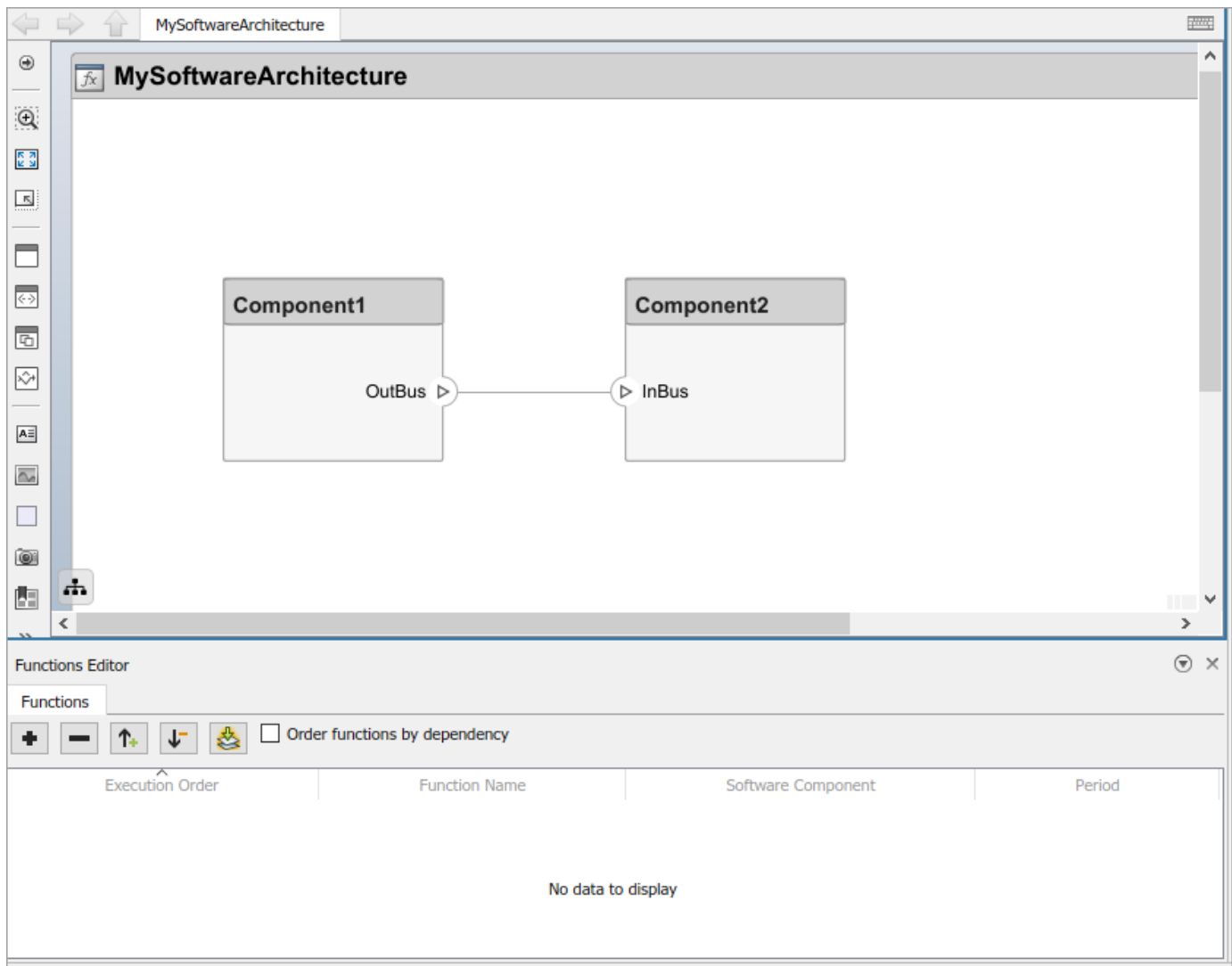
- Author and visualize functions.
- Implement behaviors for the functions.
- Import and export functions.
- Add custom properties to functions using stereotypes.

Author and Visualize Functions Using Functions Editor

You can author and visualize functions for your software architectures using the **Functions Editor**. The **Functions Editor** is visible only when you model software architectures.

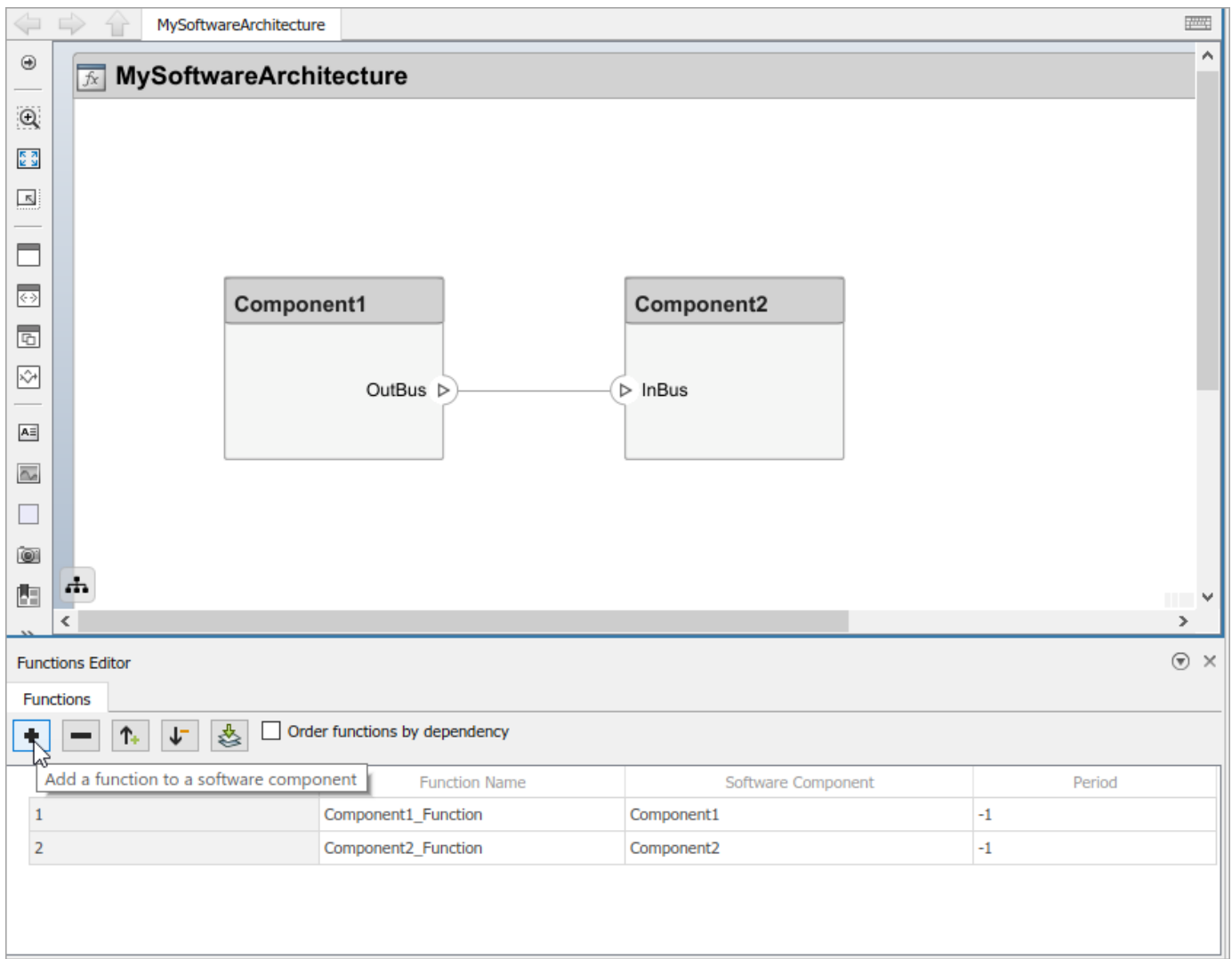
To open the **Functions Editor**, in the toolbar, navigate to **Modeling > Functions Editor**. The model automatically updates, and the table displays the functions of components in your model.

This example shows a software architecture with two components and the **Functions Editor** with an empty table.

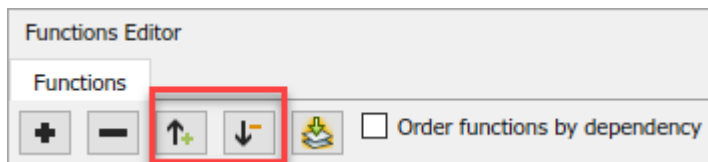


To author functions and sort them based on the order of execution:

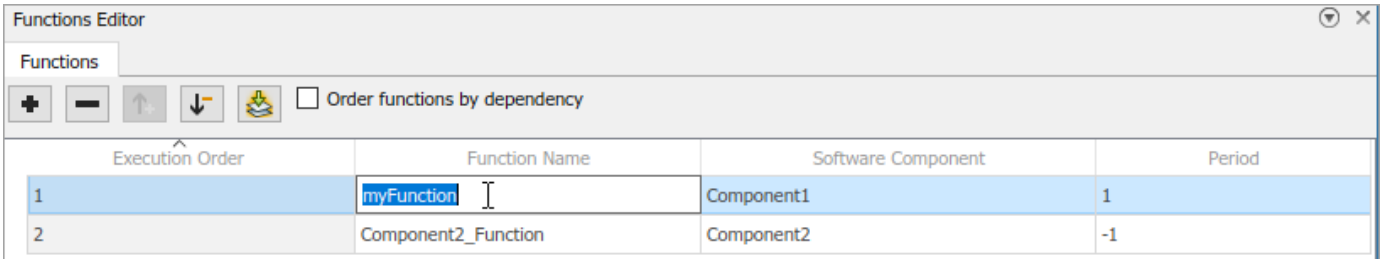
- 1 Add a function. Select Component1 as the parent. Use the same steps to add a function for Component2.



- 2 Arrange the execution order of the functions by using the up and down arrows or clicking and dragging functions to sort them.

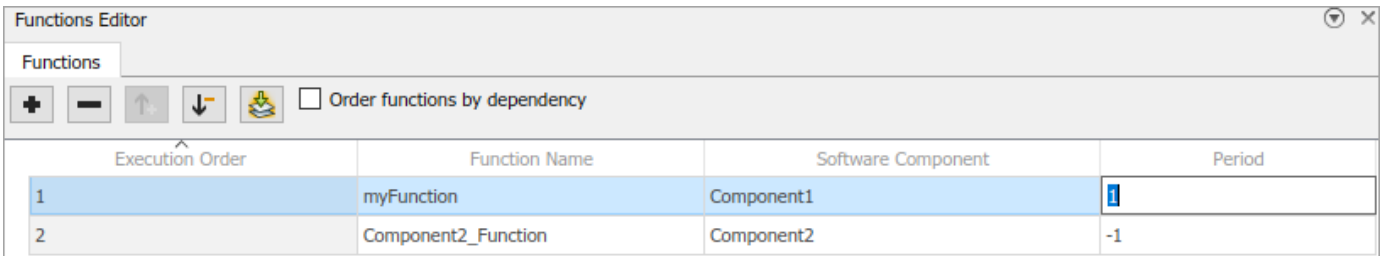


- 3 You can change the name of these functions by modifying the name in the table. Change the name of the first function to myFunction.



Execution Order	Function Name	Software Component	Period
1	myFunction	Component1	1
2	Component2_Function	Component2	-1

- 4 You can edit sample times of these functions by specifying their period in the table. Change the period of the first function to 1.



Execution Order	Function Name	Software Component	Period
1	myFunction	Component1	1
2	Component2_Function	Component2	-1

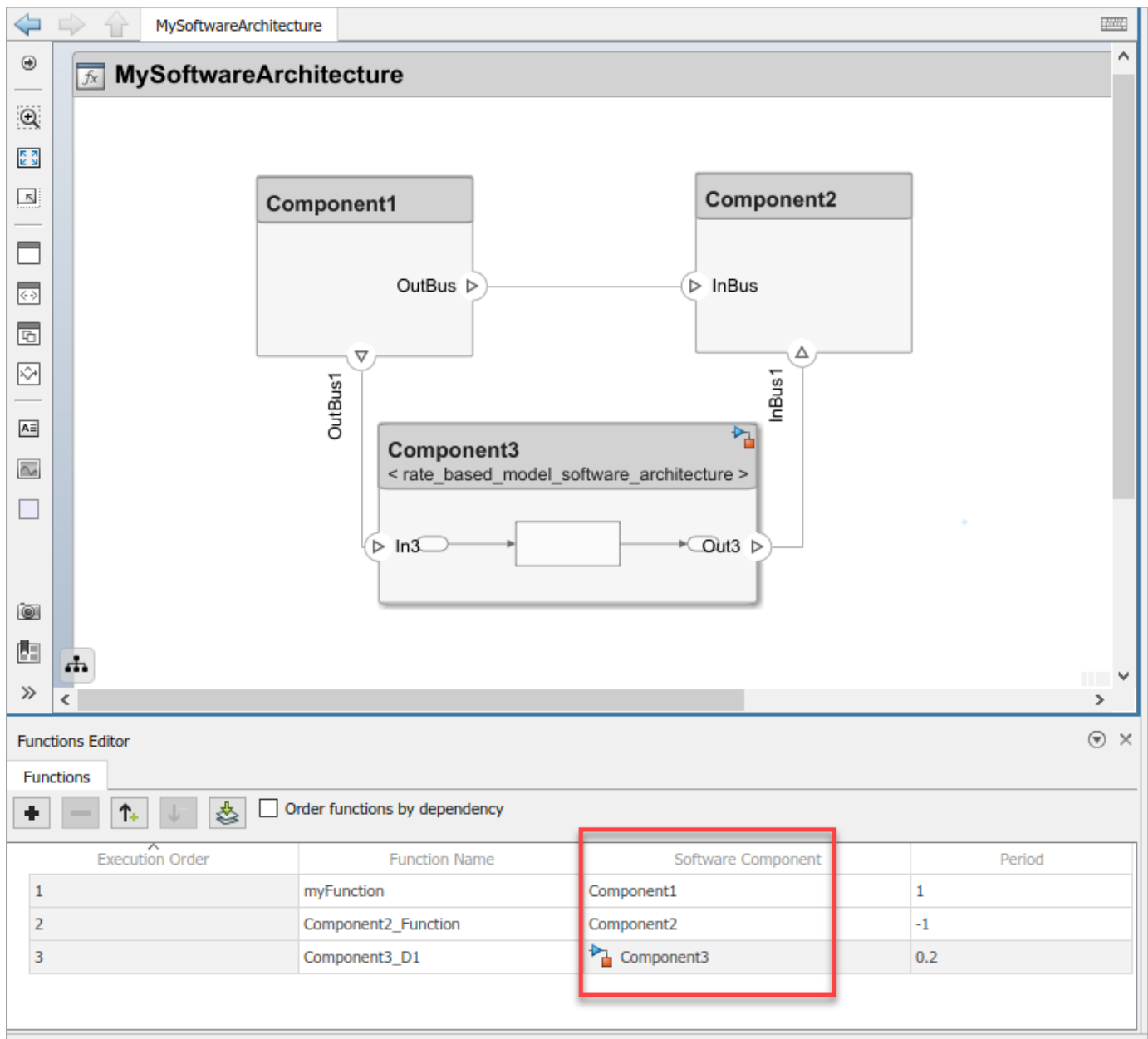
- 5 You can order functions automatically based on their data dependencies. This functionality is available for functions from behavior models. To enable automatic sorting, select the **Order functions by dependency** check box or enable `OrderFunctionsByDependency` on the architecture model.

```
set_param('RateBasedExportFunctionSoftwareArchitectureModel','OrderFunctionsByDependency','on')
```

The default value for the parameter is off.

The **Functions Editor** visualizes the functions created at the architecture level and the functions implemented in a Simulink model that is referenced by a component.

In this example, a third function is created in a Simulink behavior model, and the model is referenced by a third component, `Component3`. The **Software Component** column of the table shows the difference between functions created at the architecture level and functions created in a Simulink behavior and referenced by a component.



Author Functions Programmatically

You can also author functions for your components using the `addFunction` function.

Use the `addFunction` function to add a set of functions to the software architecture component, architecture with specified names `functionNames`.

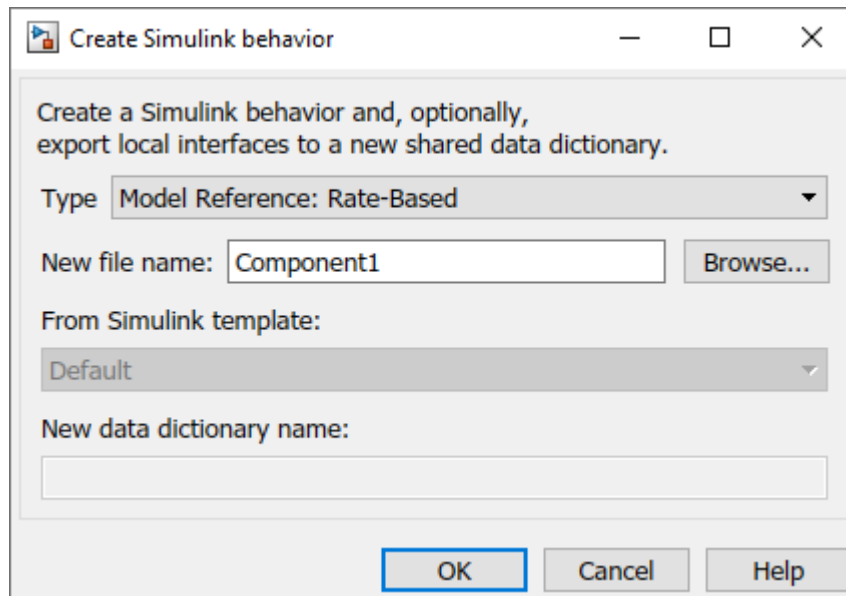
```
addFunction(architecture,functionNames)
```

For more information, see `addFunction`.

Implement Behaviors for Functions in the Architecture Level

You can create functions in the architecture level, and then implement behaviors for your functions.

- To implement functions using the toolstrip:
 - 1 Under the **Modeling** tab, select **Component**, and select **Create Simulink Behavior**.
 - 2 Select the **Type** of the Simulink behavior as rate-based or export-function.



Alternatively, you can right-click a component and select **Create Simulink Behavior**.

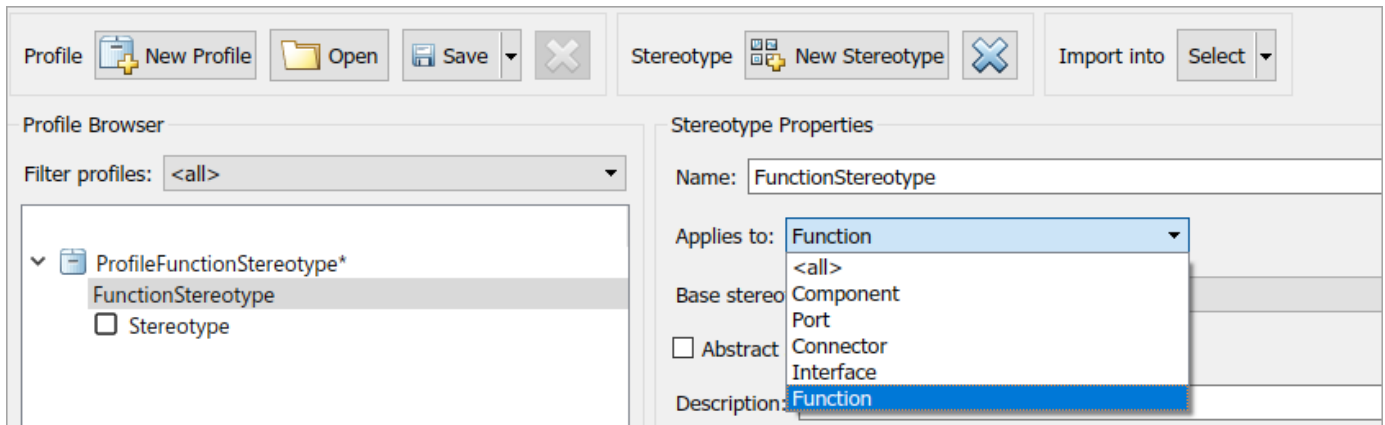
- You can also use the `createSimulinkBehavior` function to implement functions programmatically. The function creates a new rate-based or export-function behavior and links the software component to the new model. You can create rate-based or export-function behaviors only for software architectures.

```
createSimulinkBehavior(component, "mySoftwareModel", BehaviorType="RateBased")
```

Apply Stereotypes to Functions of Software Architectures

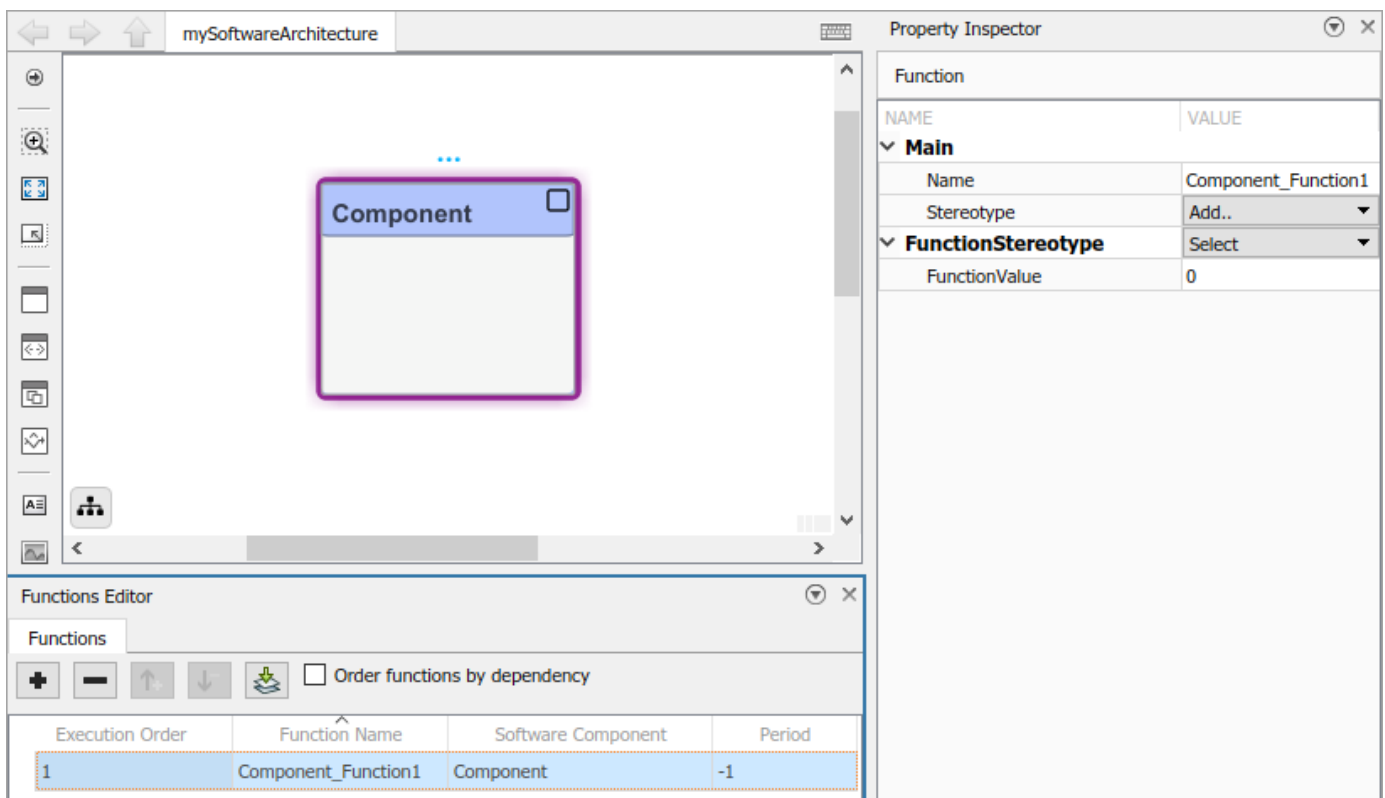
You can extend software architecture functions by adding stereotypes containing custom properties. These steps describe how to add stereotypes to your functions and are very similar to the steps to add stereotypes to other architectural elements. For more information, see “Extend Architectural Elements”.

- 1 Define your function stereotypes using the **Profile Editor**.



- 2 Use the **Functions Editor** to select functions in your software component, apply stereotypes, view the stereotypes applied to your functions, and edit the stereotype property values.

In this example, you can specify the value for the `FunctionValue` property of the stereotype called `FunctionStereotype` using the Property Inspector.



Import and Export Functions of Software Architectures

You can import and export functions of your software architectures.

- Use the `systemcomposer.exportModel` function to output a `functions` field that contains a table with information such as the name, execution order, parent component ID, period, and stereotypes of a function.

This example shows how to export a software architecture model `mySoftwareArchitecture`. The `exportedSet` output has the `functions` field that contains the table with function information.

```
exportedSet = systemcomposer.exportModel('MySoftwareArchitecture')
```

```
exportedSet =
```

```
  struct with fields:
```

```
      components: [4x5 table]
      ports: [6x4 table]
      connections: [3x5 table]
      portInterfaces: [0x9 table]
      requirementLinks: [0x15 table]
      domain: 'Software'
      functions: [3x4 table]
```

```
>> exportedSet.functions
```

```
ans =
```

```
  3x4 table
```

Name	ExecutionOrder	CompID	Period
"myFunction"	"1"	"1"	"1"
"Component2_Function"	"2"	"2"	"-1"
"Component3_D1"	"3"	"3"	"0.2"

- Use the `systemcomposer.importModel` function to import a model with functions where the `importStruct` argument can have a `functions` field that contains function information.

See Also

More About

- “Author Software Architectures” on page 8-2
- “Simulate and Deploy Software Architectures” on page 8-8
- “Modeling the Software Architecture of a Throttle Position Control System” on page 8-14
- “Display Component Hierarchy and Architecture Hierarchy Using Views” on page 9-21

Merge Message Lines Using Adapter Block

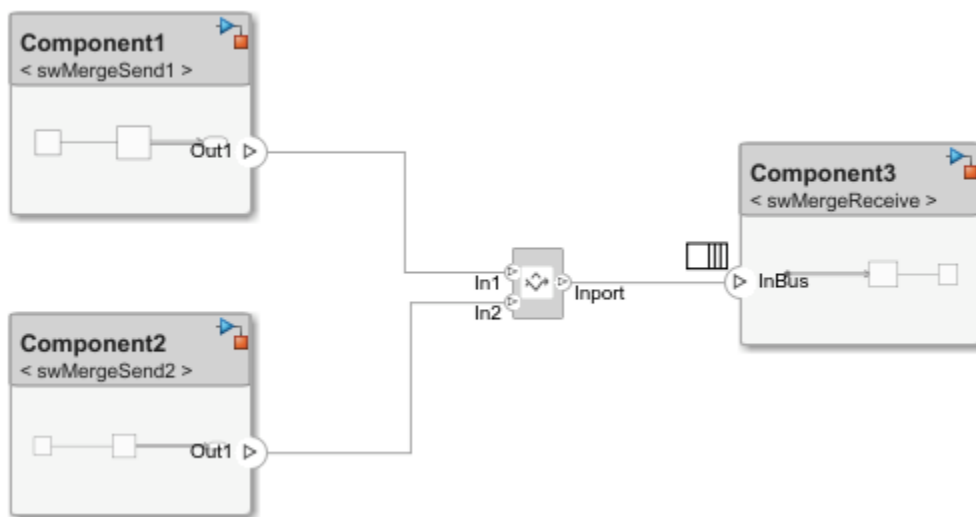
This example shows how to use a Merge block to route messages between software components in a software architecture. A Merge block is an Adapter block preconfigured to merge message and signal lines.

Open the Model

Open the model.

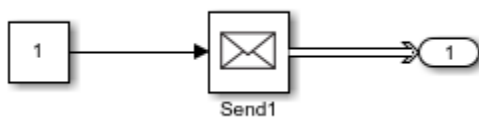
```
systemcomposer.openModel('MergeMessagesfromSoftwareComponents');
```

In this model, message-based communication is constructed between three software components: two send components, Component1 and Component2 create messages and send them to a receive component, Component3.

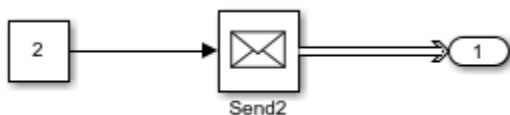


A FIFO queue is used as a message buffer between the components.

Component1 is linked to the Simulink® behavior model `swMergeSend1` that generates messages with value 1 in every 0.1 sample time.



Component2 is linked to the Simulink behavior `swMergeSend2` that generates messages with value 2 in every 0.3 sample time.

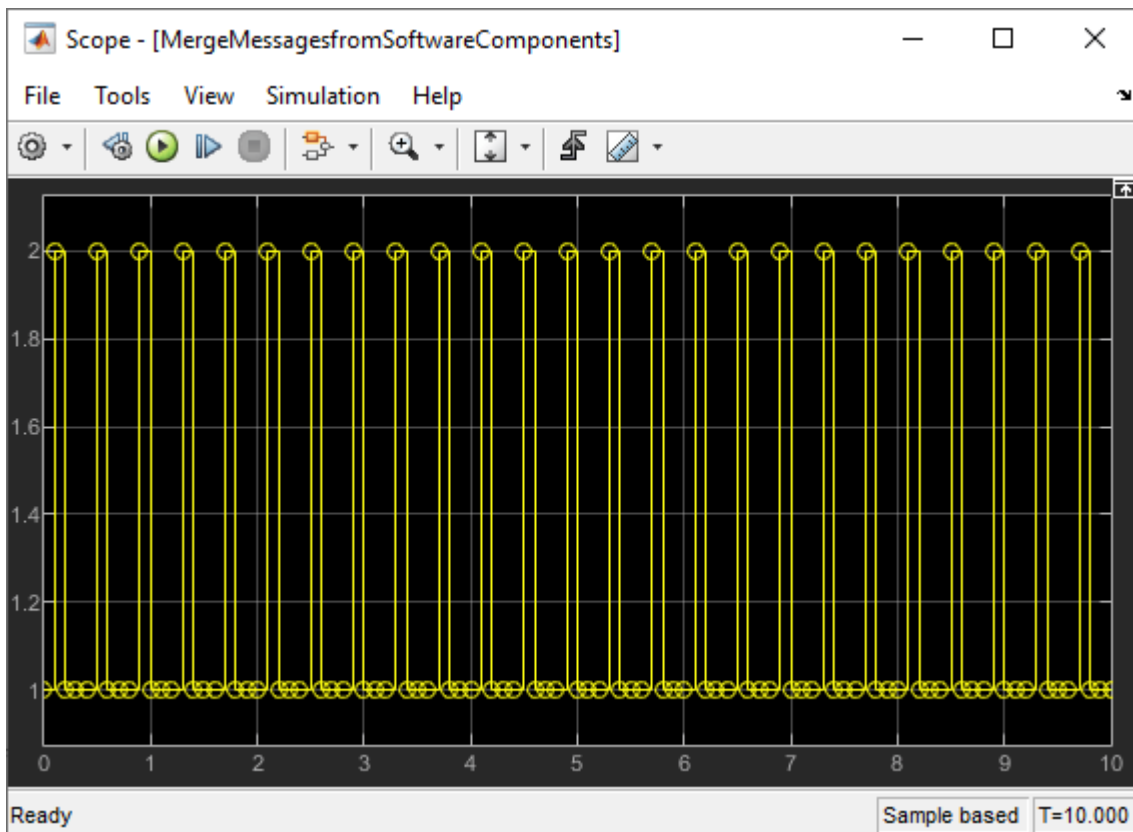


Component3 is linked to the Simulink behavior `swMergeReceive` that receives messages and converts them to signals. The In Bus Element port block is used to configure the queue outside the component as a FIFO queue of capacity 100.



Simulate Model and Review Results

Simulate the model. Observe that the Scope block in `swMergeReceive` displays the values received from both components.



See Also

Adapter

More About

- "Author Software Architectures" on page 8-2
- "Simulate and Deploy Software Architectures" on page 8-8

Authoring Functions for Software Components of an Adaptive Cruise Control

This example shows how to design an adaptive cruise control system in System Composer™:

- Capture the system architecture in terms of constituent software components and functions.
- Indicate which functions are critical for maintaining the desired speed of the vehicle using custom properties.
- Implement the behavior of the functions using Simulink.

Adaptive Cruise Control Architecture

An adaptive cruise control (ACC) system is a control system that modifies the speed of a vehicle in response to conditions on the road. As in regular cruise control, the driver sets a desired speed for the vehicle. Additionally, the adaptive cruise control system can slow the vehicle down if there is another vehicle moving more slowly in the lane in front of it.

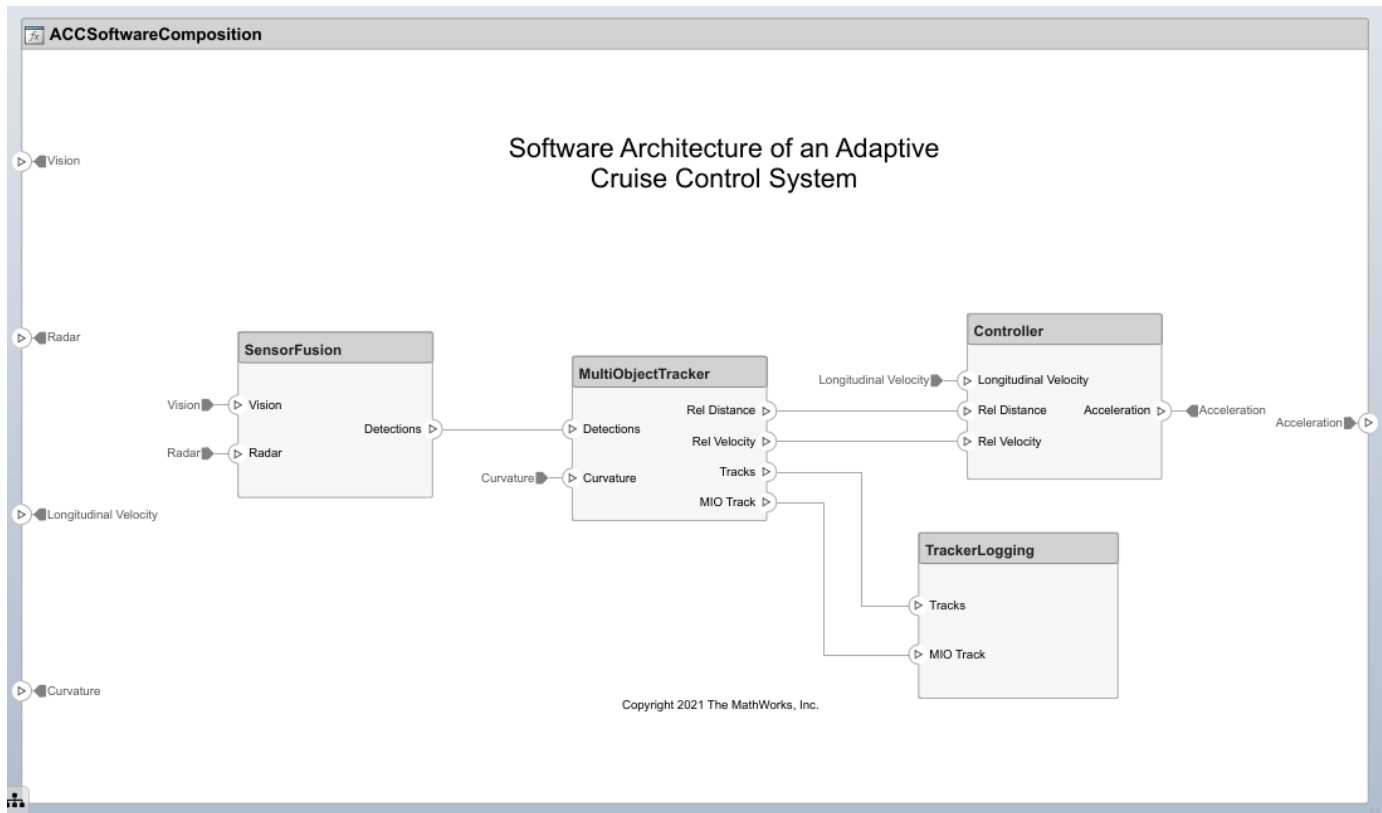
The ACC algorithm requires that the vehicle knows the curvature of the road, the relative distance, and velocity of the lead vehicle immediately in front of it. For more information about the ACC algorithm and the entire system, see “Adaptive Cruise Control with Sensor Fusion” (Automated Driving Toolbox). The system software retrieves detections from a radar sensor and video from a camera, which are fused by a sensor fusion component for more accurate detections. The detections are provided to multi-object tracker to determine the relative distance and velocity of the lead vehicle. These states and the longitudinal velocity of the vehicle are then provided to a controller component to compute the acceleration to apply to the vehicle to maintain a safe distance.

Author Components and Interfaces

First, create the architecture of the adaptive cruise control software. To open a new software architecture model, use this command.

```
systemcomposer.createModel('ACCSoftwareCompositionScratch', 'SoftwareArchitecture', true);
```

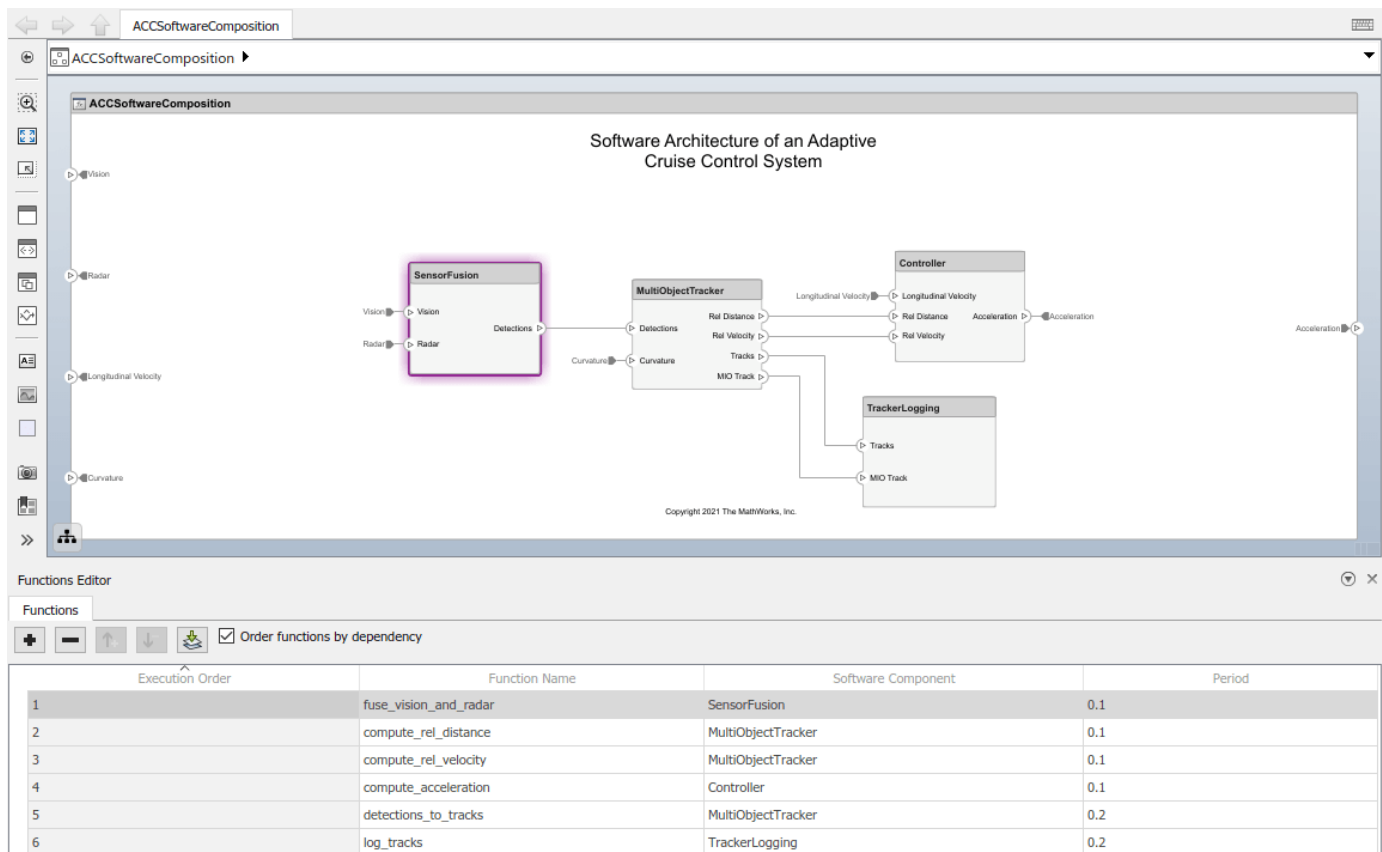
The system is composed of a `SensorFusion` component, a `MultiObjectTracking` component, a `Controller` component, and a `TrackerLogging` component for monitoring.



Author Component Functions

To specify the functions that define the behavior of each component, open the Functions Editor. To open the Functions Editor, on the **Modeling** tab, in the **Design** section, click **Functions Editor**. To create the functions to implement the behavior of each component, select a component and use the add button. For more information about authoring functions using the Functions Editor, see “Author and Extend Functions for Software Architectures” on page 8-24.

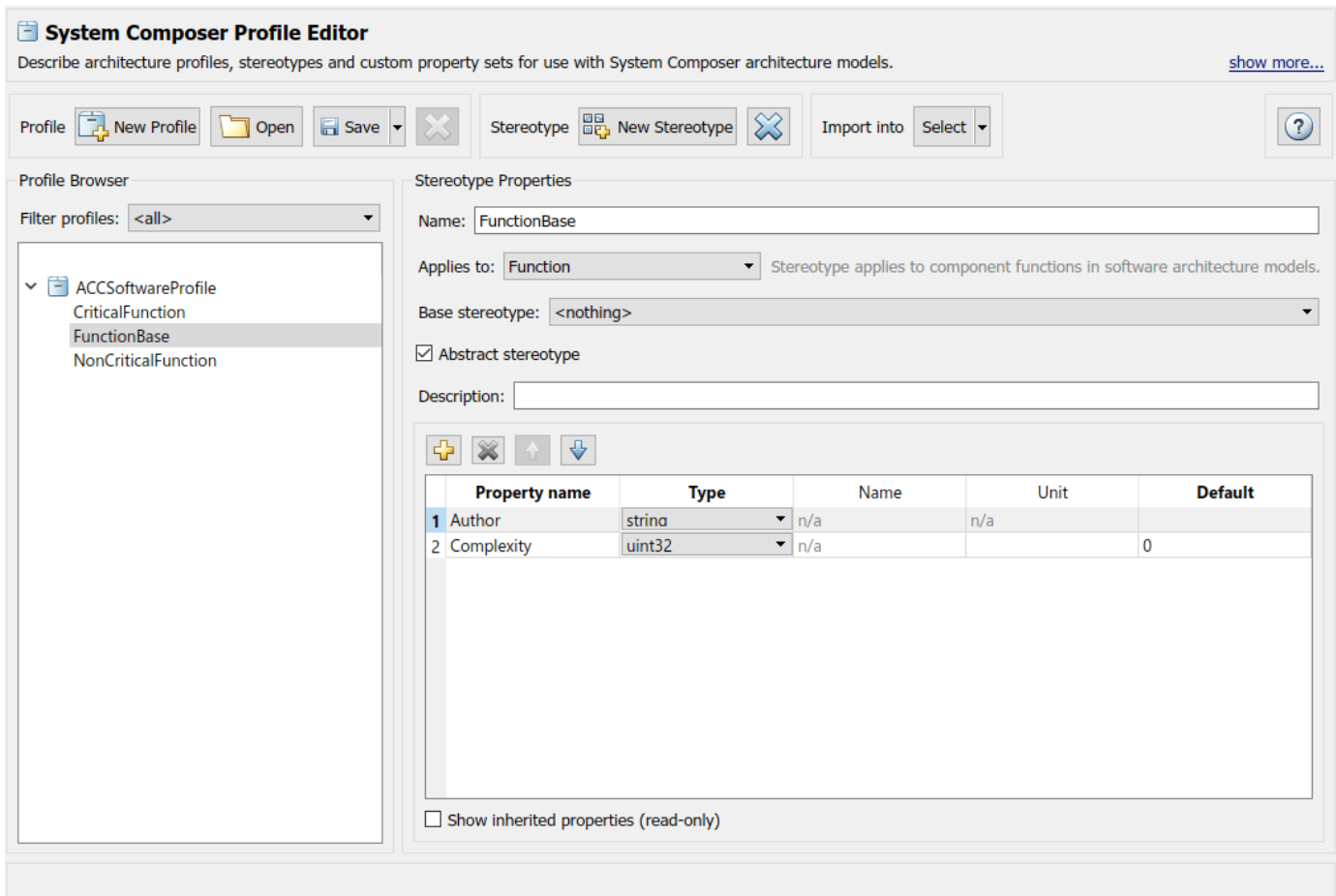
You can modify the built-in function properties such as the name, period, or execution order. The name and period properties can be modified by editing the corresponding cells in the table. You can specify the execution order of functions in the Functions Editor by dragging functions into the desired order or by selecting a function and clicking the up and down buttons.



Add Custom Properties to Functions

You can apply custom properties using System Composer profiles and stereotypes. Load the profile `ACCSoftwareProfile` and import it into the composition. The profile contains three stereotypes.

- `FunctionBase` is a stereotype used as the base for all function stereotypes.
- `CriticalFunction` stereotype applies to functions that are critical in determining the output acceleration.
- `NonCriticalFunction` stereotype applies to functions that are not critical in determining the output acceleration.



Add custom properties to a function by applying stereotypes from the loaded profile.

- 1 To open the Property Inspector, select **Modeling > Design > Property Inspector**.
- 2 In the Functions Editor, select `fuse_vision_and_radar`.
- 3 In the Property Inspector, select **Stereotype > Add** `ACCSoftwareProfile.CriticalFunction` to apply the stereotype.

Copyright 2021 The MathWorks, Inc.

Functions Editor

Functions

Order functions by dependency

Execution Order	Function Name	Software Component	Period
1	fuse_vision_and_radar	SensorFusion	0.1
2	compute_rel_distance	MultiObjectTracker	0.1
3	compute_rel_velocity	MultiObjectTracker	0.1
4	compute_acceleration	Controller	0.1
5	detections_to_tracks	MultiObjectTracker	0.2
6	log_tracks	TrackerLogging	0.2

This stereotype designates functions that are executed to determine the output acceleration. In the ACC software architecture, all functions are critical to determining the acceleration except for the functions defined in the TrackerLogging component.

Generate Code for Functions

Code for the adaptive cruise control system can be generated and deployed to the target hardware using Embedded Coder®. To generate code, execute the `slbuild` command, or press **Ctrl+B** to build the model.

```
slbuild('ACCSoftwareCompositionScratch');
```

```

← → ACCSoftwareComposition.cpp 🔍 Search
18 #include "ACCSoftwareComposition.h"
19
20 // Model step function for TID1
21 void ACCSoftwareComposition::fuse_vision_and_radar() // Explicit Task: fuse_vision_and_radar
22 {
23     // RootInportFunctionCallGenerator generated from: '<Root>/fuse_vision_and_radar'
24 }
25
26 // Model step function for TID2
27 void ACCSoftwareComposition::compute_rel_distance() // Explicit Task: compute_rel_distance
28 {
29     // RootInportFunctionCallGenerator generated from: '<Root>/compute_rel_distance'
30 }
31
32 // Model step function for TID3
33 void ACCSoftwareComposition::compute_rel_velocity() // Explicit Task: compute_rel_velocity
34 {
35     // RootInportFunctionCallGenerator generated from: '<Root>/compute_rel_velocity'
36 }
37
38 // Model step function for TID4
39 void ACCSoftwareComposition::compute_acceleration() // Explicit Task: compute_acceleration
40 {
41     // RootInportFunctionCallGenerator generated from: '<Root>/compute_acceleration'
42 }
43
44 // Model step function for TID5
45 void ACCSoftwareComposition::detections_to_tracks() // Explicit Task: detections_to_tracks
46 {
47     // RootInportFunctionCallGenerator generated from: '<Root>/detections_to_tracks'
48 }
49

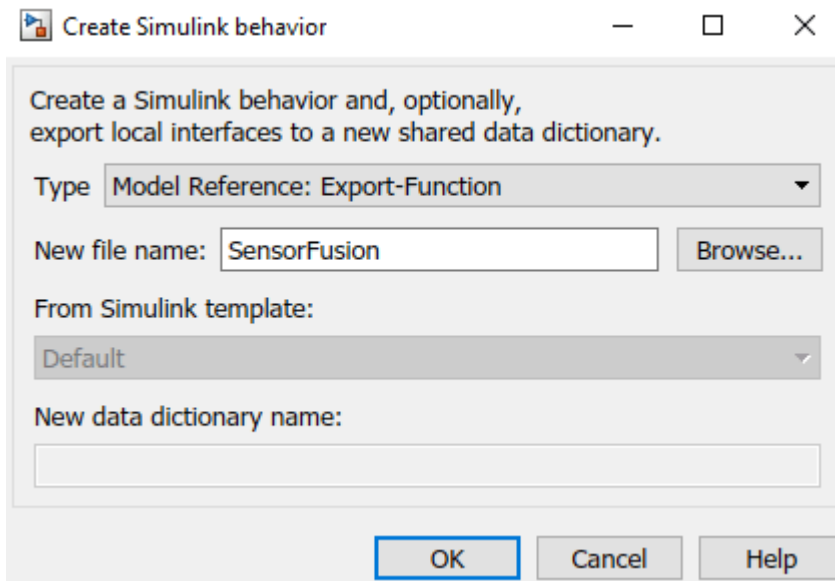
```

Since no component has a linked behavior, the generated code contains empty definitions for each function in the software architecture.

Implement Behaviors for Functions

You can implement the behaviors for functions of a component in Simulink by creating a Simulink behavior. Right-click the SensorFusion component and select **Create Simulink Behavior**, or navigate to **Modeling > Component > Create Simulink Behavior**. To choose the type of the Simulink behavior, for **Type**, select **Model Reference: Export-Function** or **Model Reference: Rate-Based**. Click **OK** to create a SensorFusion export-function model linked to the SensorFusion component.

For more information on using the export-function modeling style, see “Export-Function Models Overview”.



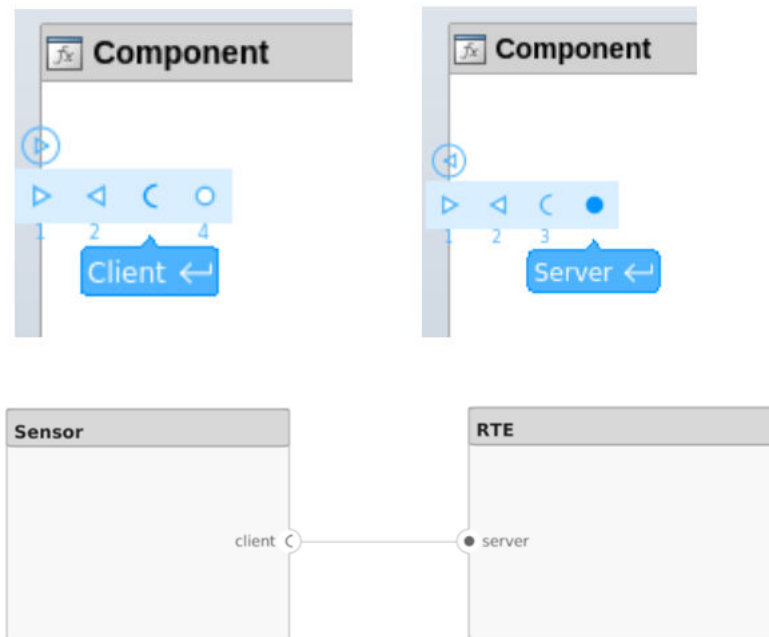
The new model contains one inport block with a function-call output port, `fuse_vision_and_radar`, with a sample time of 0.1 seconds, as specified in the Functions Editor. You can connect the output port to a function-call subsystem that models the behavior of that function.

Copyright 2021 The MathWorks, Inc.

Author Service Interfaces for Client-Server Communication

You can model client-server connections between software components in software architectures in System Composer using client ports and server ports and associating service interfaces with these ports.

To expose services performed by a software component, create a server port on that component. To access those services from within another software component, create a client port on the second component and connect the two ports. The ball and socket icons represent server and client ports, respectively. You can also author client and server ports at the composition level, with client-server lines crossing multiple hierarchies.



A service interface defines the functional interface between client and server components. Each service interface consists of one or more function elements. Use the **Interface Editor** to author and edit service interfaces.

Interfaces

Composite Data Interface
Add data interface

Value Type
Add value type

Physical Interface
Add physical interface

Service Interface
Add service interface

Search Dictionary View

	Dimensions	Units	Complexity	
	1		real	□
	1		real	□

Interfaces

myModel.slx

ServiceInte

		Dimensions	Units	Complexity
y = f0(u)				
u	double	1		real
y	double	1		real
y = f1(u)				
u	double	1		real
y	double	1		real

Edit the function signature to change the number and names of inputs and outputs. Edit function element properties as you would edit other interface element properties. Function argument types can include built-in types as well as bus objects.

Interfaces

myModel.slx

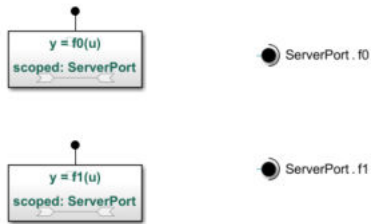
ServiceInterface0

	Type	Dimensions	Units	Complexity
[y1,y2] = f0(u1,u2)				
u	double	1		real
y	double	1		real

Once you have defined a service interface in the **Interface Editor**, you can assign it to client and server ports using the Property Inspector. You can also use the Property Inspector to assign stereotypes to service interfaces.

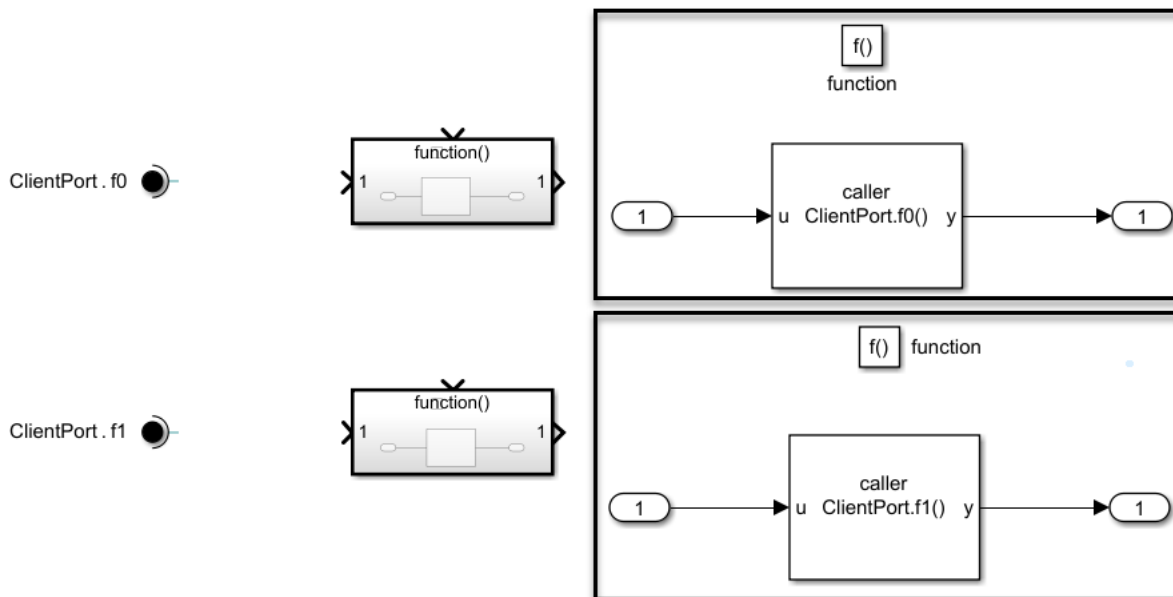
To implement function behavior for components with client or server ports using referenced Simulink models, right-click a component and select **Create Simulink behavior**, or use the `createSimulinkBehavior` function. System Composer creates a new export-function model and links the component to the new model.

For a component with a server port, the model contains a Function Element block and a port-scoped Simulink Function block for each function element of the service interface associated with the server port. The Simulink Function blocks are preconfigured with a function interface specification to match each function element of the service interface. The Function Element block creates an exporting function port in the Simulink model. The attributes of the port are based on the service interface definition.



To implement the desired algorithm for each server function, open the Simulink Function blocks and add and connect the needed blocks and other modeling elements.

For a component with a client port, the model contains a Function Element Call block and a Function-Call Subsystem block containing a Function Caller block for each function element of the service interface associated with the client port. The Function Caller blocks are preconfigured with a **Function prototype** and argument specifications to match each function element of the service interface. The Function Element Call block creates an invoking function port in the Simulink model. The attributes of the port are based on the service interface definition.



See Also

Function Element | Function Element Call | Simulink Function | Function Caller

Related Examples

- “Define Port Interfaces Between Components” on page 3-2
- “Call Simulink Functions in Other Models Using Function Ports”
- “Service-Oriented Sensor Modeling” on page 8-44
- “Software Component Modeling”

Service-Oriented Sensor Modeling

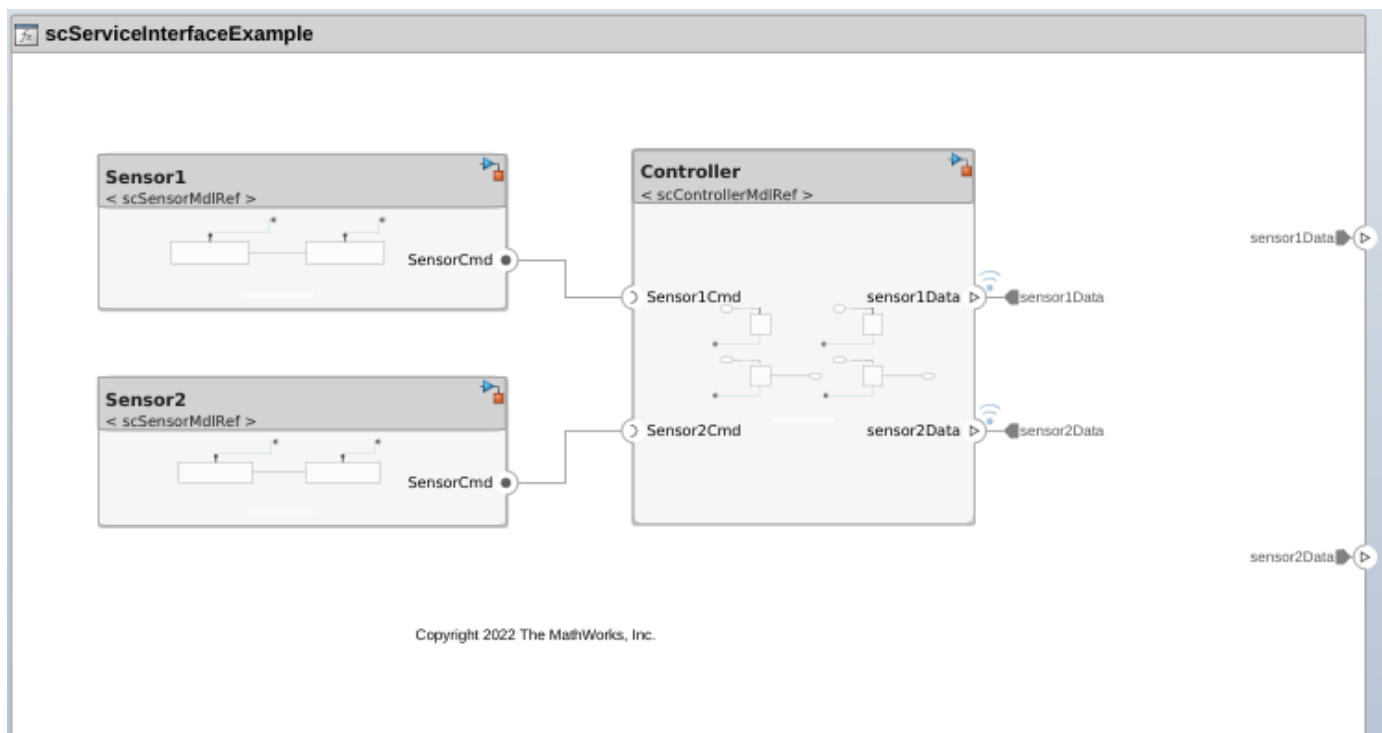
This example shows how to use a service interface in a software architecture model to allow a component to call services provided by specific instances of a referenced component.

Overview

In this example, the model `slexServiceInterfaceExample` consists of a controller component, `Controller`, and two sensor components, `Sensor1` and `Sensor2`. The sensor components are modeled as two different instances of the same referenced model, `scSensorModelRef`. The referenced model defines two services: `reset`, which resets the sensor from drifting over time, and `fetchData`, which reads the latest sensor values. A single service interface is specified between the controller and the two sensor instances, which allows the controller to call `reset` or `fetchData` for a specific instance of the referenced sensor component.

Open the model.

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



Both instances of the referenced sensor model output a sine wave with different amplitudes. You can view and specify the amplitudes in the Model Data Editor of the top model. To access the Model Data Editor, go to the **Modeling** tab, and in the **Design** section, select **Model Data Editor**.

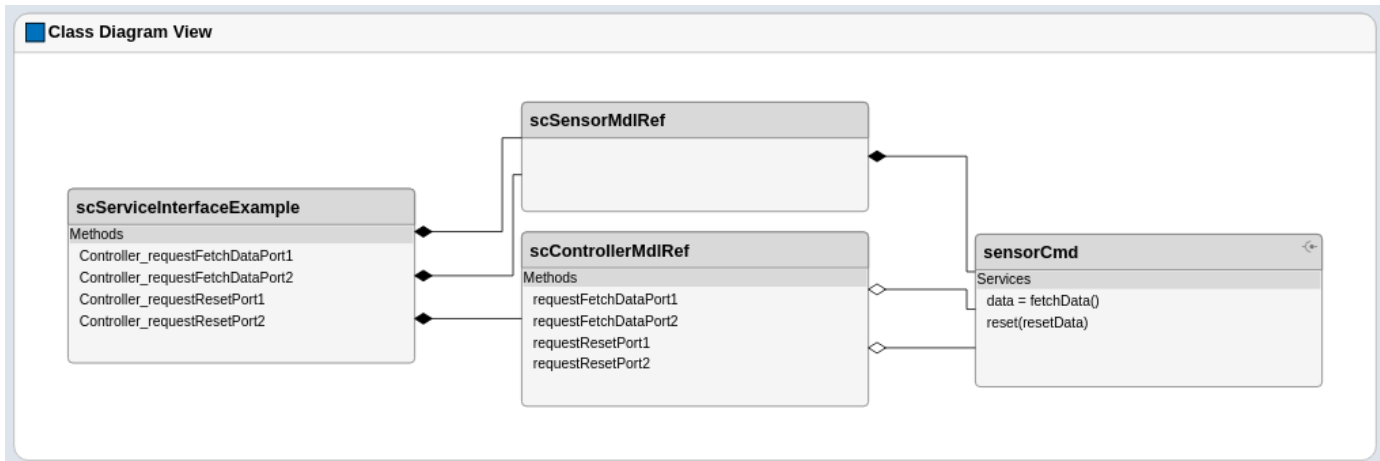
Model Data Editor

Inports/Outports | Signals | Data Stores | States | Parameters

Design

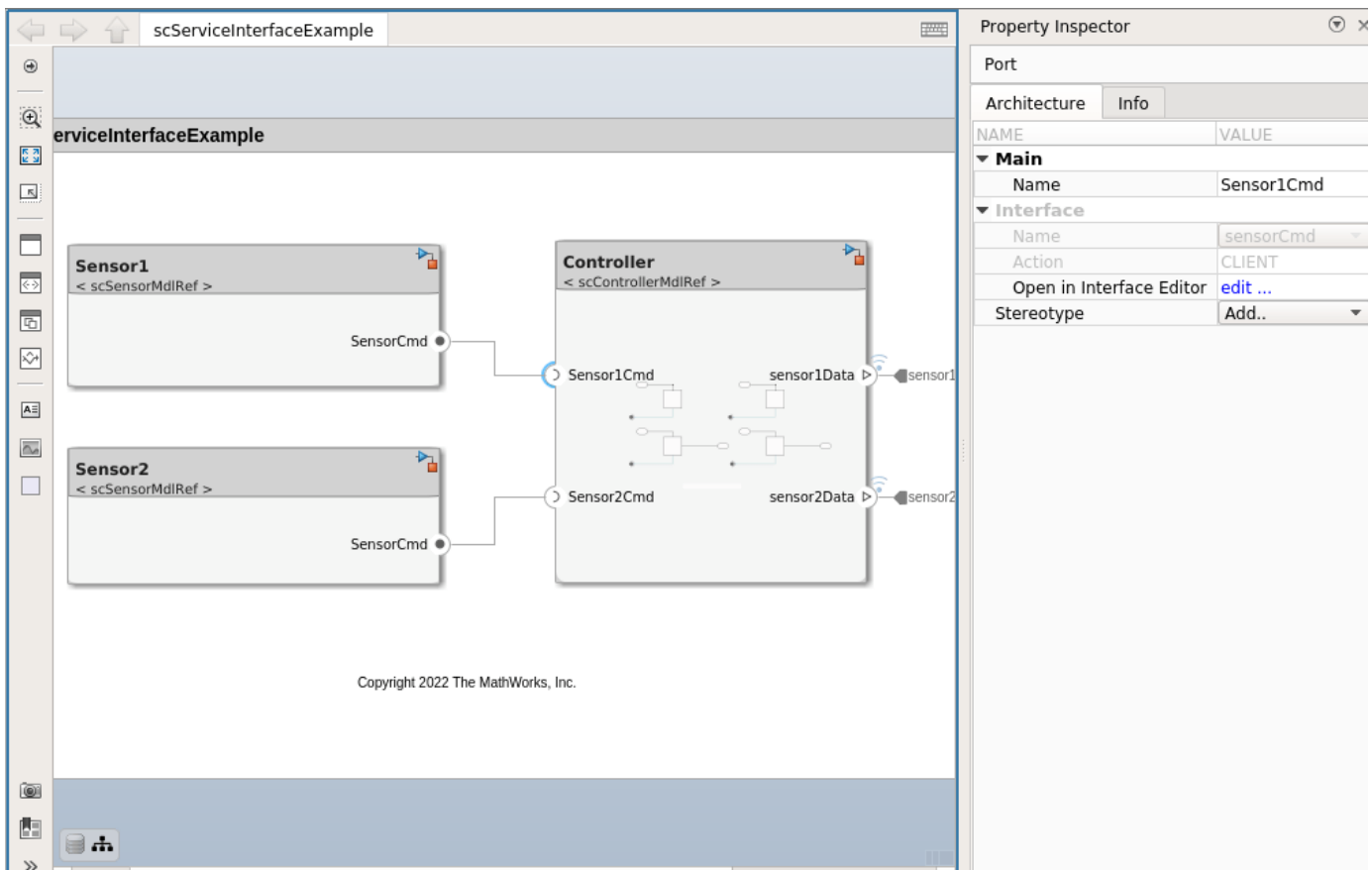
Source	Name	Value	Data Type	Min	Max	Dimensions	Unit	Argument
Sensor1	Amplitude	1						<input type="checkbox"/>
Sensor2	Amplitude	2						<input type="checkbox"/>

A Class Diagram View can be used to visualize the relationship between the controller and the referenced sensor component. To open the Class Diagram View, go to the **Modeling** tab, and in the **Views** section, select **Architecture Views**.



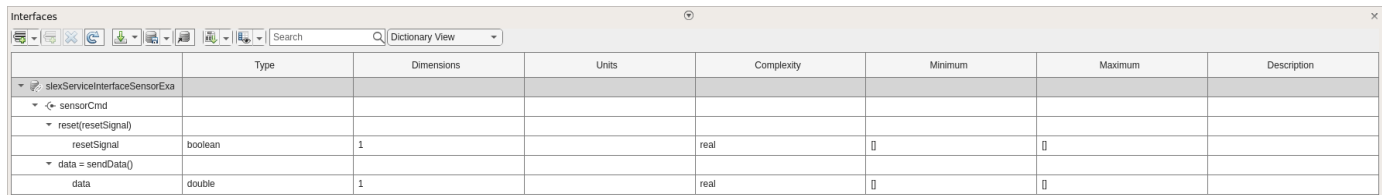
Client-Server Ports

The controller component interacts with the sensor components using client-server ports. Function calls to the services provided by the sensors and their responses are handled through these ports. Each client-server port is mapped to the service interface `sensorCmd`, which defines the services.



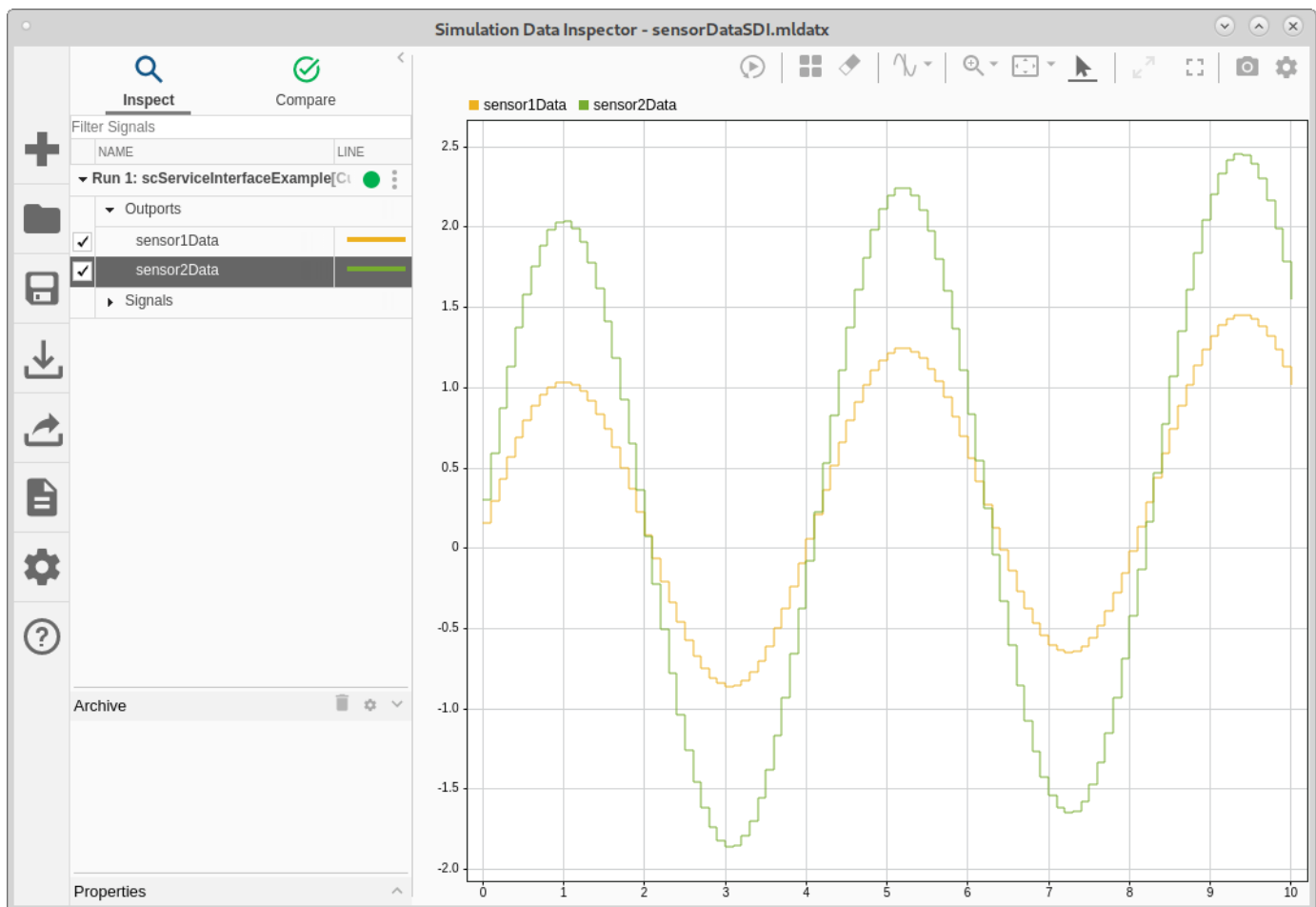
Service Interface Specification

The service interface `sensorCmd` is defined through the Interface Editor. To access the Interface Editor, go to the **Modeling** tab, and in the **Design** section, select **Interface Editor**. The service interface `sensorCmd` is used across referenced models and stored in the data dictionary `slxServiceInterfaceExample.sldd`. Note that `sensorCmd` contains two functions, `reset` and `fetchData`.



	Type	Dimensions	Units	Complexity	Minimum	Maximum	Description
slxServiceInterfaceSensorExa							
sensorCmd							
reset(resetSignal)	boolean	1		real	0	0	
data = sendData()	double	1		real	0	0	

In this example, the sensors exhibit a drift that increases over time. The controller calls the `fetchData` service to receive the sensor output and calls the `reset` service periodically to reset the drift back to 0. To visualize the sensor outputs, go to the **Simulation** tab, and in the **Review Results** section, select **Data Inspector**.



Function Scheduling

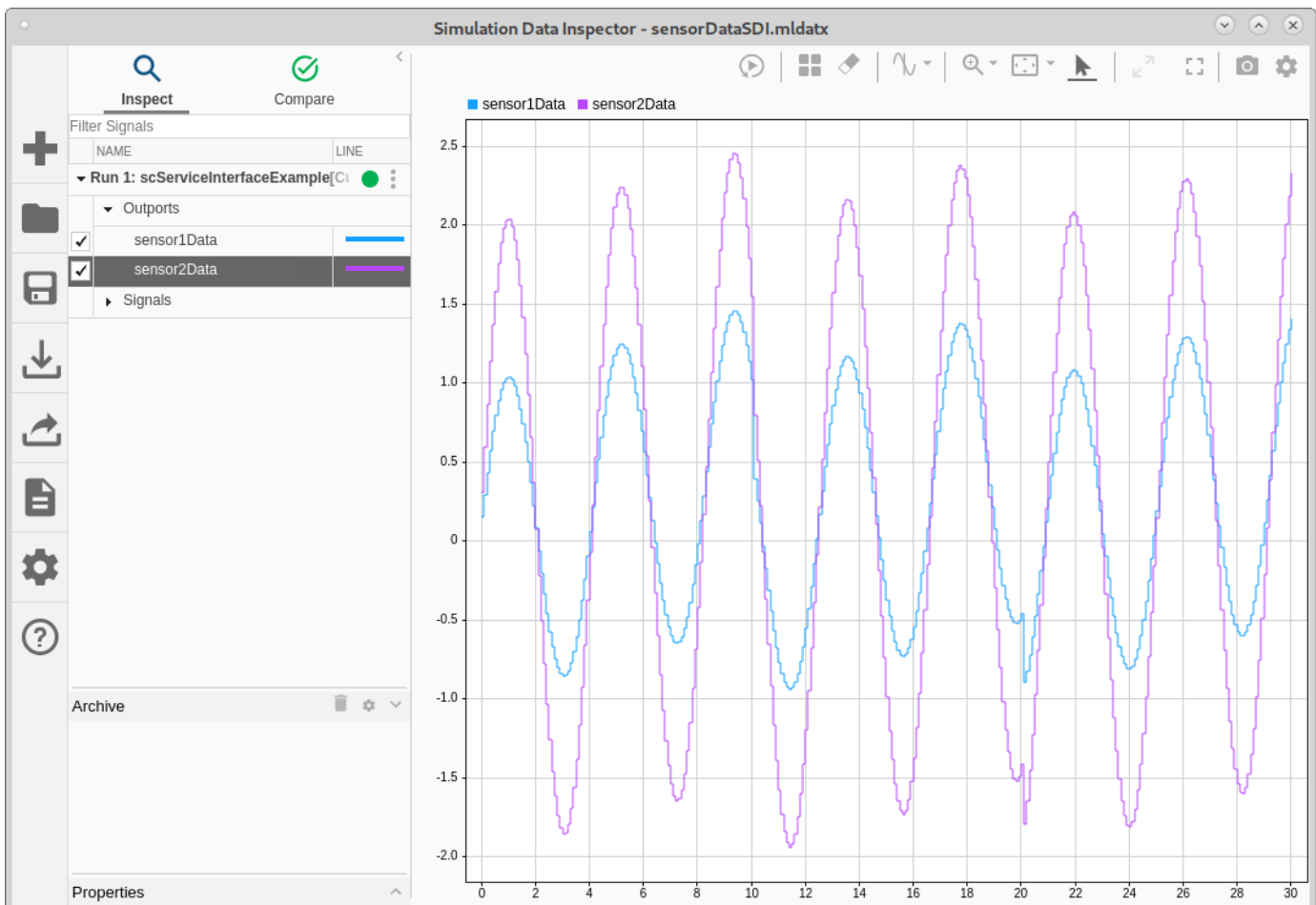
You can schedule the functions of the controller through the Functions Editor. To access the Functions Editor, go to the **Modeling** tab, and in the **Design** section, select **Functions Editor**. In this example, the `reset` function is called by the controller every 0.2 seconds, while the `fetchData` function is called every 0.1 seconds. Note that there is a Boolean input argument for the `reset` function, `resetData`. By default, `resetData` is `false`, and thus the `reset` function does not reset the sensors until the controller sets `resetData` to `true`. In this example, the controller component is configured to set `resetData` to `true` every 50 samples of the `reset` function, or every 10 seconds.

Functions Editor

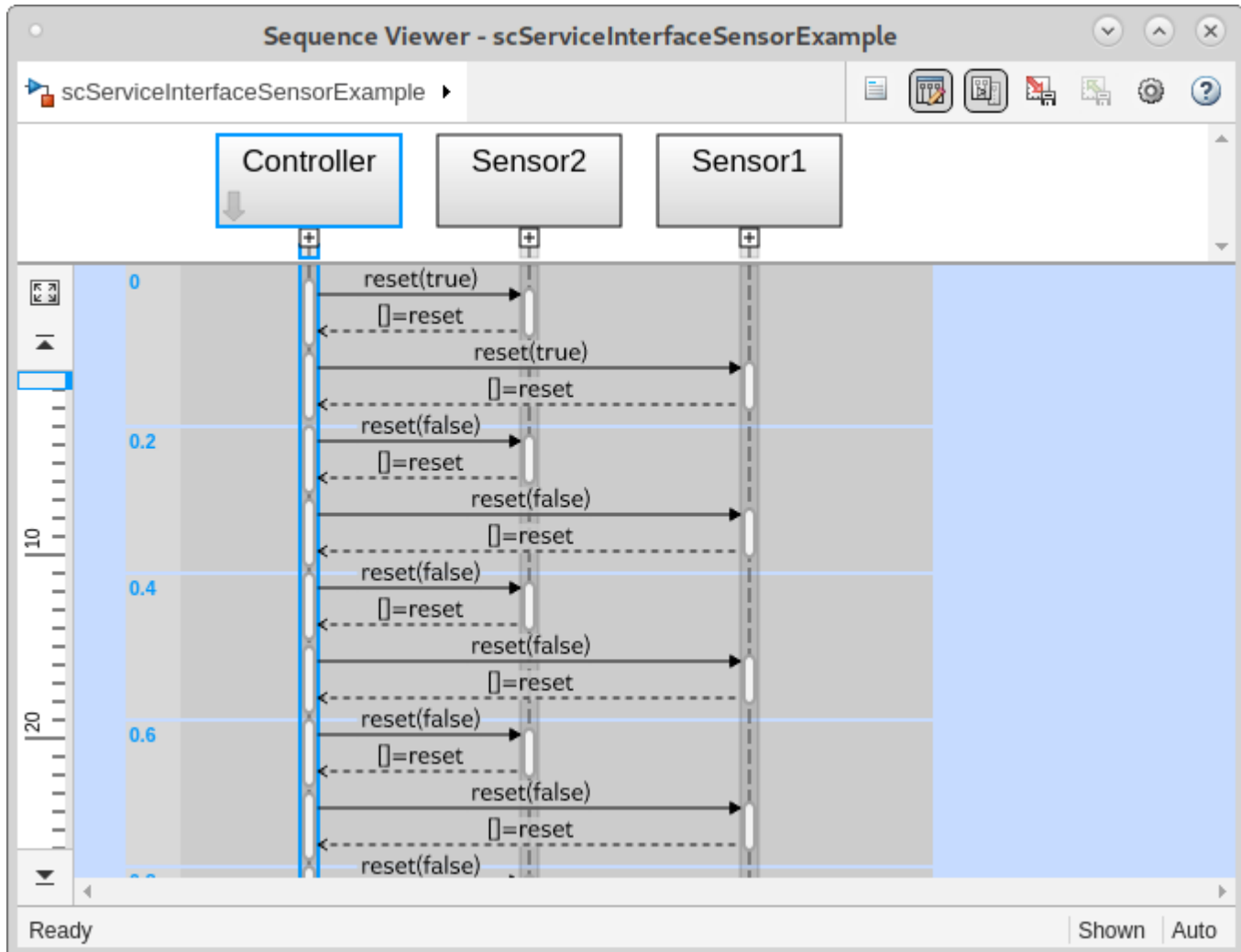
Functions

Order functions by dependency

Execution Order	Function Name	Software Component	Period
1	Controller_requestFetchDataPort1	Controller	0.1
2	Controller_requestFetchDataPort2	Controller	0.1
3	Controller_requestResetPort1	Controller	0.2
4	Controller_requestResetPort2	Controller	0.2



You can view the sequence of function calls throughout the simulation of the model in the Sequence Viewer. To access the Sequence Viewer, go to the **Simulation** tab, and in the **Review Results** section, select **Sequence Viewer**.



Code Generation

To generate code for the model, which includes the service interface `sensorCmd`, use this command.

```
rtwbuild('scServiceInterfaceSensorExample');
```

Note that the service interface `SensorCmd` is generated as an abstract class. This action enables implementation to be separate from the interface.

```
#ifndef RTW_HEADER_sensorCmdT_h_
#define RTW_HEADER_sensorCmdT_h_
#include "rtwtypes.h"

class sensorCmdT
{
public:
    virtual void reset(boolean_T resetData)
    {
    }

    virtual ~sensorCmdT()
    {
    }

    virtual void fetchData(real_T *data)
    {
    }
};

#endif // RTW_HEADER_sensorCmdT_h_
```

This abstract class is implemented by the generated code for the referenced sensor model.

```

// Output and update for referenced model: 'scSensorMdlRef'
void scSensorMdlRef::fetchData(real_T *rty_data)
{
    // Outputs for Function Call SubSystem: '<Root>/fetchData'
    // MATLAB Function: '<S1>/MATLAB Function' incorporates:
    //   SignalConversion generated from: '<S1>/reset'

    if (scSensorMdlRef_B.TmpSignalConversionAtresetData0) {
        scSensorMdlRef_DW.offset = 0.0;
    } else {
        scSensorMdlRef_DW.offset += 0.005;
    }

    scSensorMdlRef_DW.time += 0.1;

    // SignalConversion generated from: '<S1>/data' incorporates:
    //   MATLAB Function: '<S1>/MATLAB Function'

    *rty_data = std::sin(1.5 * scSensorMdlRef_DW.time) *
        *scSensorMdlRef_rtp_Amplitude + scSensorMdlRef_DW.offset;

    // End of Outputs for SubSystem: '<Root>/fetchData'
}

// Output and update for referenced model: 'scSensorMdlRef'
void scSensorMdlRef::reset(boolean_T rtu_resetData)
{
    // Outputs for Function Call SubSystem: '<Root>/reset'
    // SignalConversion generated from: '<S2>/resetData'
    scSensorMdlRef_B.TmpSignalConversionAtresetData0 = rtu_resetData;

    // End of Outputs for SubSystem: '<Root>/reset'
}

```

The generated abstract class is also used to construct the class of the controller.

```

// Constructor
scControllerMdlRef::scControllerMdlRef(sensorCmdT& Sensor1Cmd_arg, sensorCmdT&
    Sensor2Cmd_arg) :
    scControllerMdlRef_DW(),
    Sensor1Cmd(Sensor1Cmd_arg),
    Sensor2Cmd(Sensor2Cmd_arg),
    scControllerMdlRef_M()
{
    // Currently there is no constructor body generated.
}

```

The service is subsequently called by the controller from the SensorCmd abstract class.


```
// Output and update for referenced model: 'scControllerMdlRef'  
void scControllerMdlRef::scControllerM_callfetchDataPort(real_T *rty_Out2)  
{  
  // RootInportFunctionCallGenerator generated from: '<Root>/callfetchDataPort' incorporates:  
  //   SubSystem: '<Root>/callfetchData'  
  
  // FunctionCaller: '<S1>/sendData Caller'  
  Sensor2Cmd.fetchData(rty_Out2);  
  
  // End of Outputs for RootInportFunctionCallGenerator generated from: '<Root>/callfetchDataPort'  
}
```

See Also

Sequence Viewer | Simulation Data Inspector

Related Examples

- “Author Service Interfaces for Client-Server Communication” on page 8-41

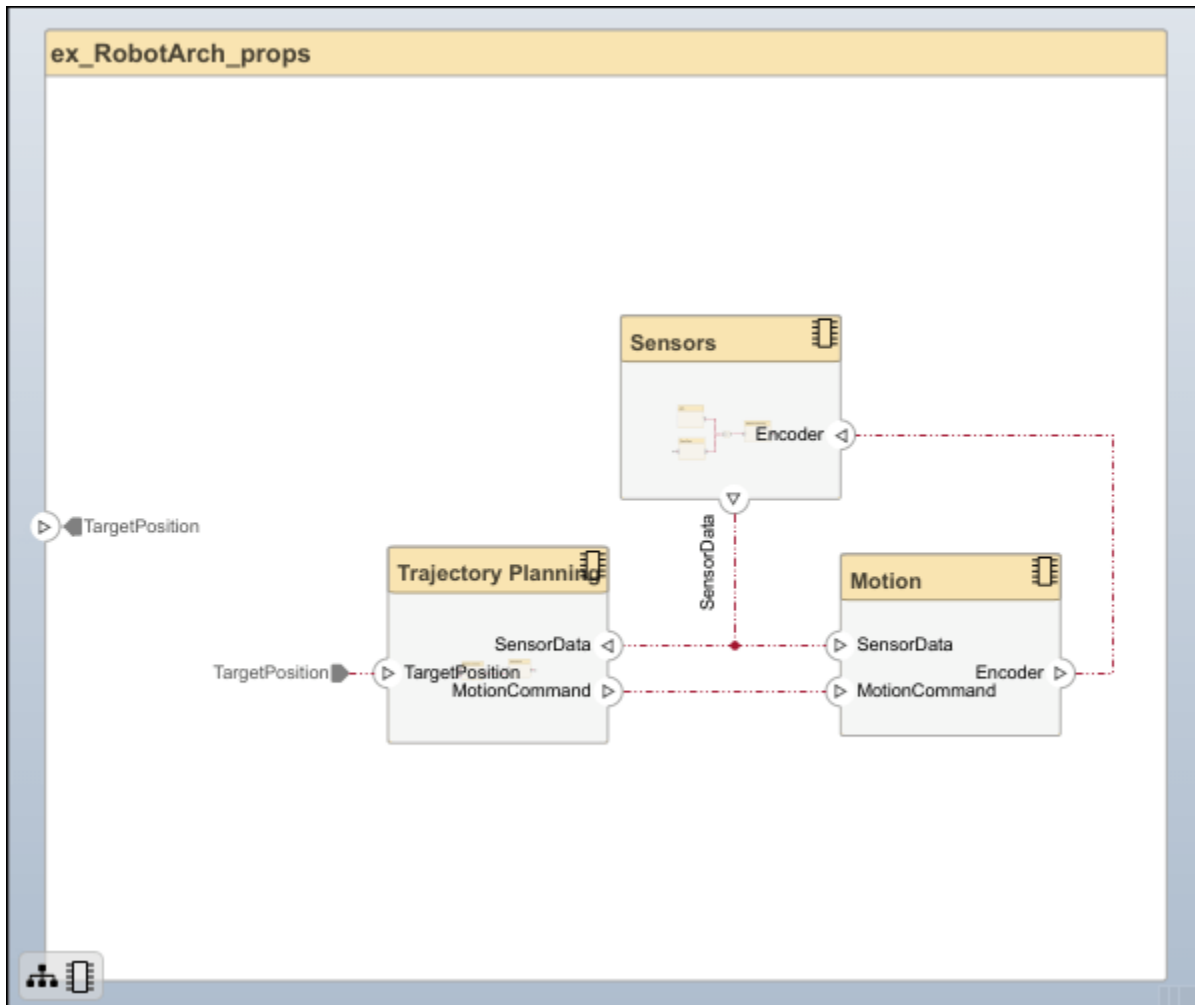
Create Custom Views

Create Spotlight Views

A system being designed in System Composer for a real application is usually 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. Use the model below to begin creating spotlight views.

Mobile Robot Architecture Model with Properties

This example shows a mobile robot architecture model with stereotypes applied to components and properties defined.



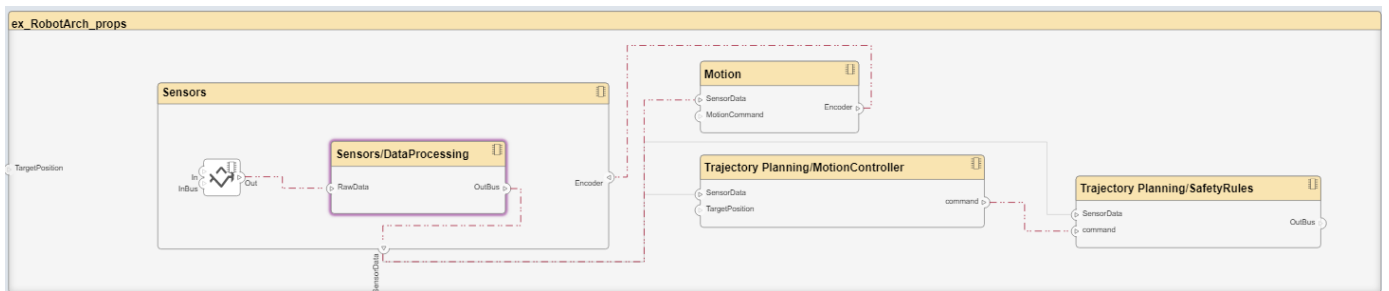
Create Spotlight Views from Components


Create views dynamically using spotlight views.

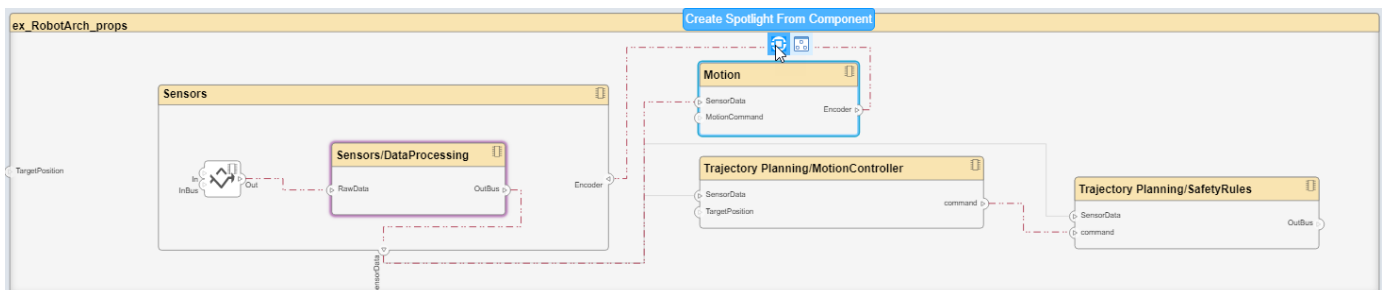
- 1 Double-click the Sensors component, then select the DataProcessing component.
- 2 Right-click the DataProcessing component and select Create Spotlight from Component. Alternatively, select the DataProcessing component and navigate to **Modeling > Architecture Views > Spotlight**.


The spotlight view launches and shows all model elements to which the DataProcessing component connects. The spotlight diagram is laid out automatically and cannot be edited. However, it allows you to inspect just a single component and study its connectivity to other components.

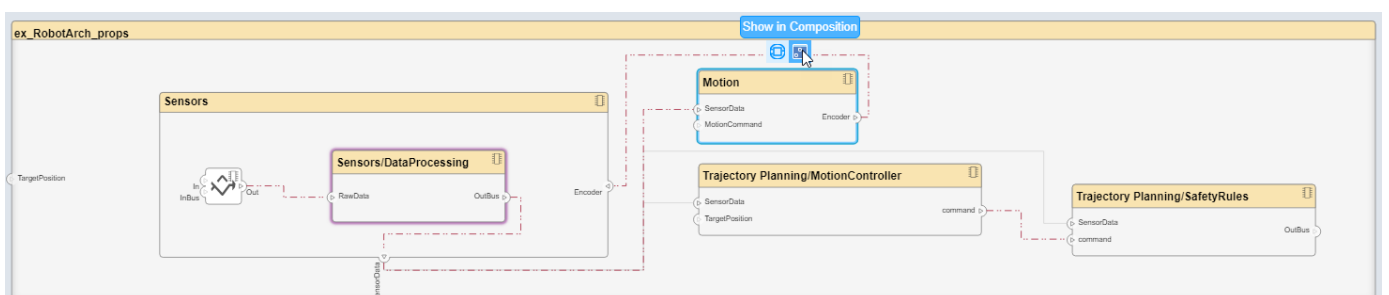
Note Spotlight views are transient. They are not saved with the model.



- 3 Shift the spotlight to another component. Select the **Motion** component. Click the ellipsis above the component to open the action menu. To create a spotlight from the component, click .



To view the architecture model at the level of a particular component, select the component and click .



- 4 To return to the architecture model view, 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. To show the spotlight view in a dedicated window, select **Open in New Window** in the component context menu, then create the spotlight view. Spotlight views are dynamic and transient: any change in the composition refreshes any open spotlight views, and spotlight views are not saved with the model.

See Also

More About

- “Create Architecture Views Interactively” on page 9-5
- “Display Component Hierarchy and Architecture Hierarchy Using Views” on page 9-21
- “Create Architectural Views Programmatically” on page 9-15
- “Modeling System Architecture of Keyless Entry System” on page 9-25

Create Architecture Views Interactively

The structural hierarchy of a system typically differs from the hierarchy of the functional requirements of a system. With architecture views in System Composer, 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. Create views by adding elements manually. Views create a simplified way to work with complex architectures by focusing on certain parts of the architectural design.

You can use different types of views to represent the system:

- *Operational views* demonstrate how a system will be used and should be 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 requirements. A view allows you to better understand what components you need to satisfy your requirements while not necessarily focusing on the structure.

This example uses the architecture model for a keyless entry system to create component diagram views.

A component diagram represents a view with components, ports, and connectors based on how the model is structured.

Component diagrams allow you to programmatically or manually add and remove components from the view.

For more information on the keyless entry system architecture, see “Modeling System Architecture of Keyless Entry System” on page 9-25.

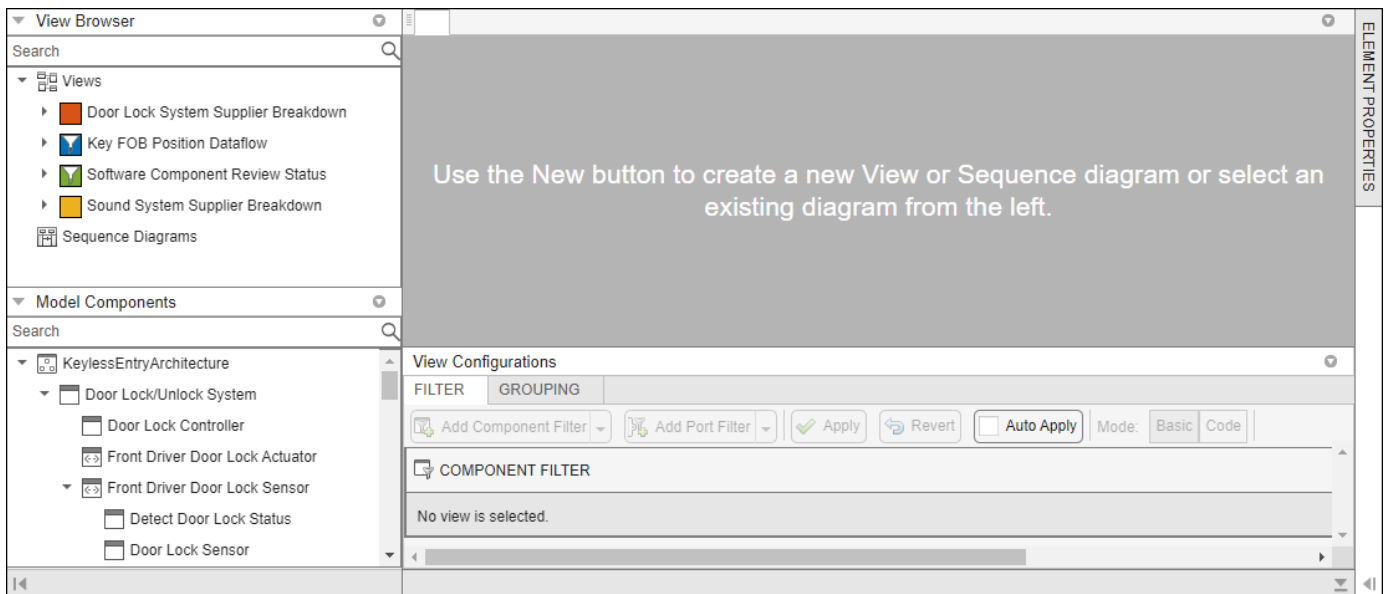
Create Filtered Views with Component Filters and Port Filters

- 1 In the MATLAB Command Window, enter this command.

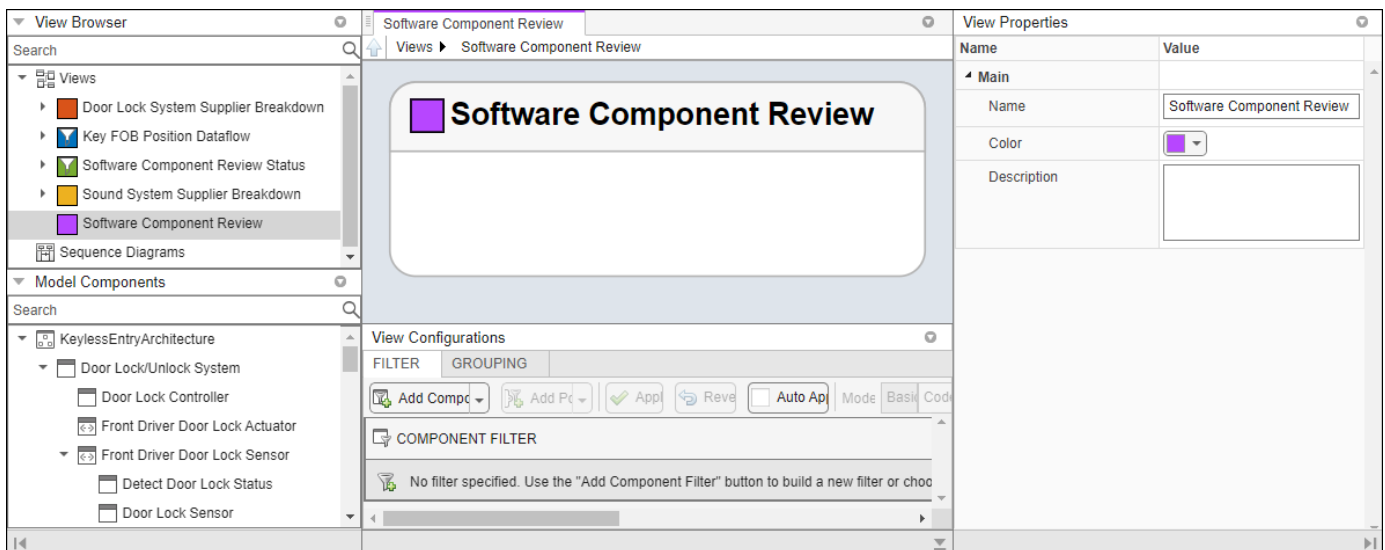
```
sckeylessEntrySystem
```

The architecture model opens in System Composer.

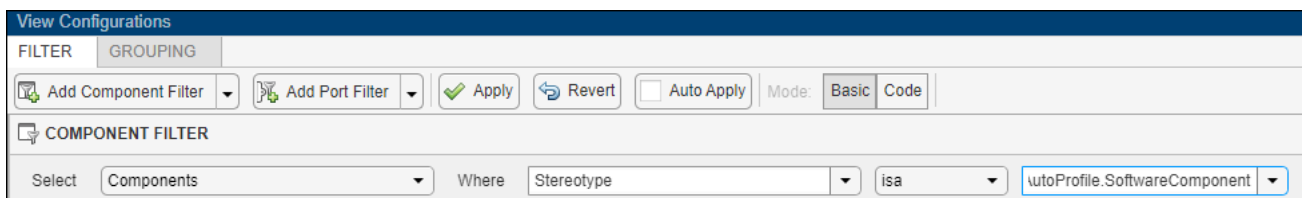
- 2 Navigate to **Modeling > Architecture Views** to open the **Architecture Views Gallery**.



- 3 Select **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.

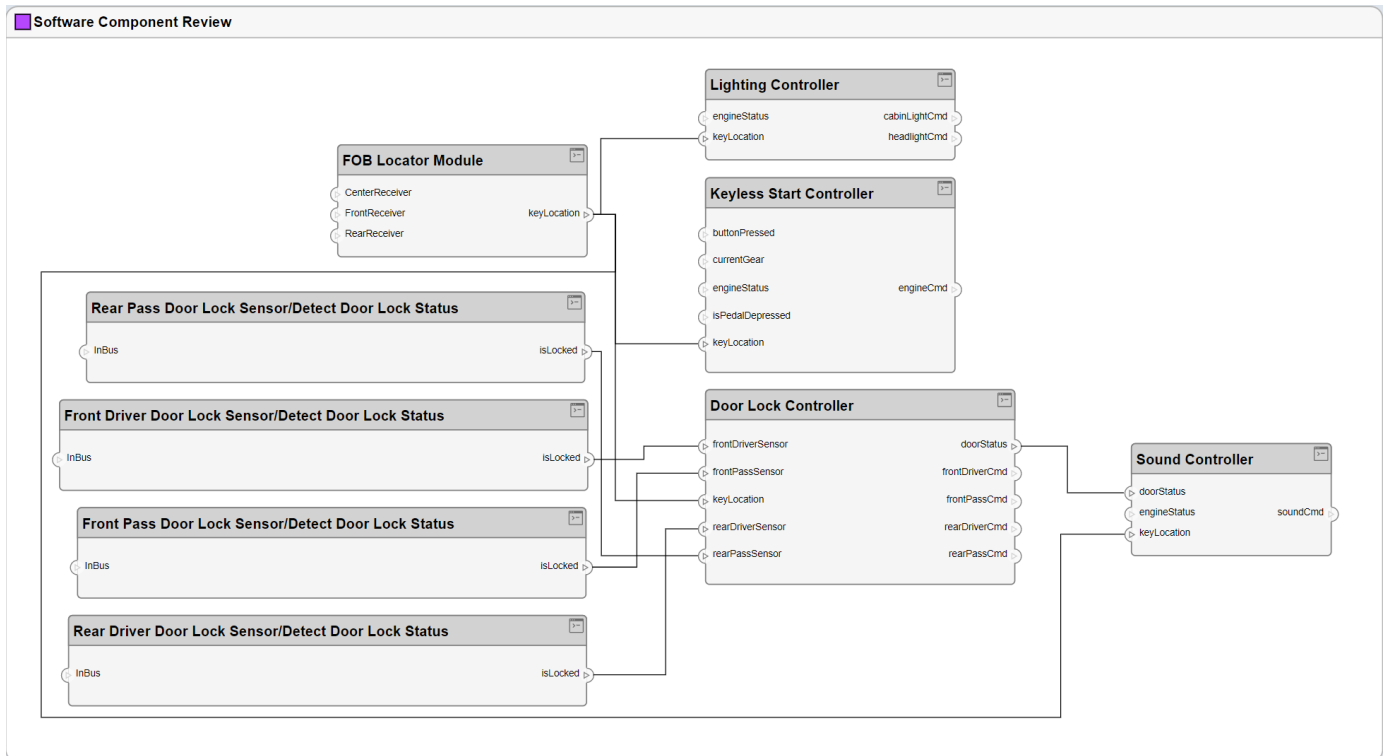


- 5 In the bottom pane, select **View Configurations > Filter > Add Component Filter** to add a form-based criterion to a component filter.
- 6 From the **Select** list, select **Components**. From the **Where** list, select **Stereotype**. Select **isa**. In the text box, from the list select **AutoProfile.SoftwareComponent**.







7 Select **Apply** .

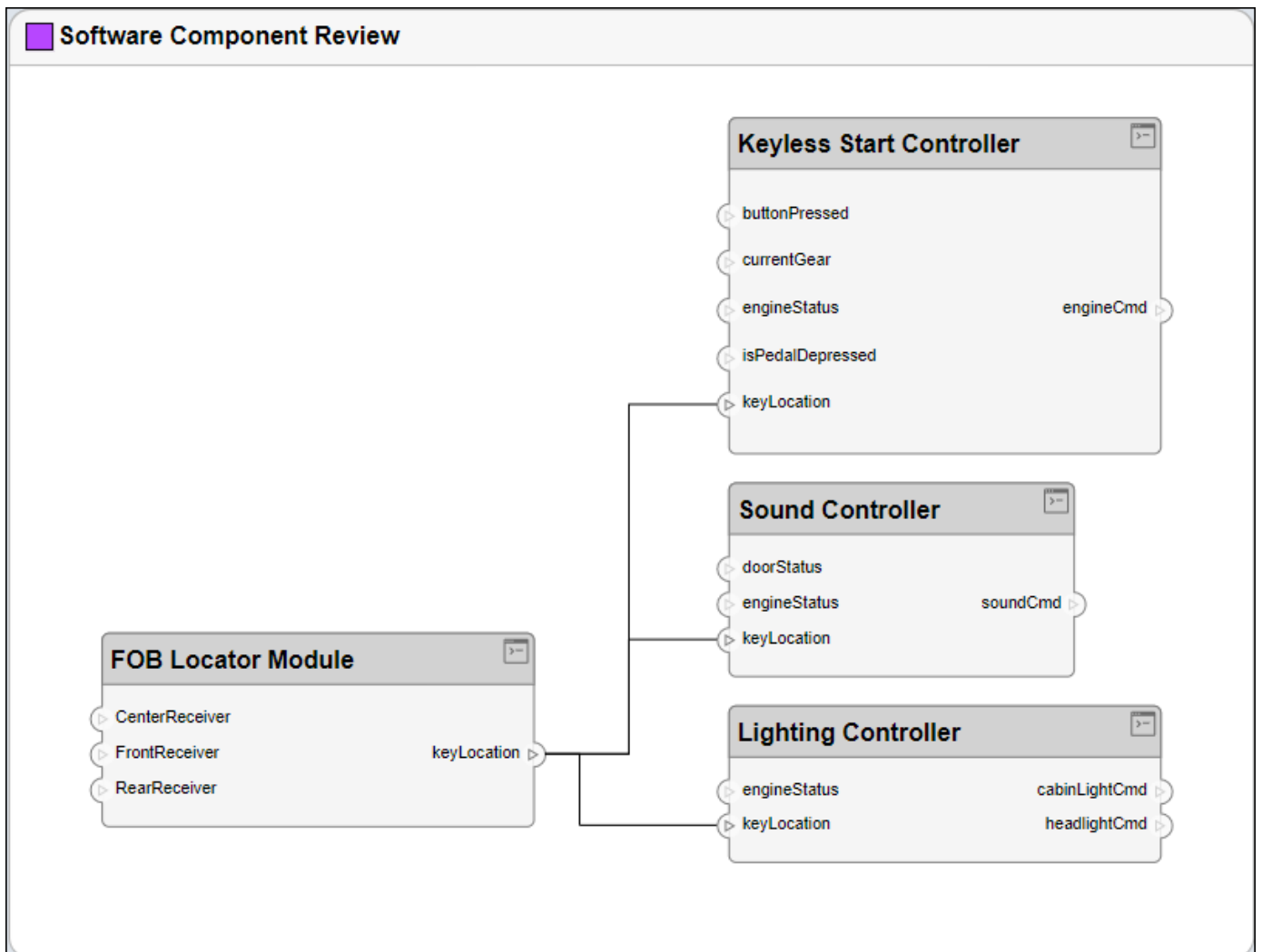
An architecture view is created using the query in the **Component Filter** box. The view is filtered to select all components with the `AutoProfile.SoftwareComponent` stereotype applied to them.



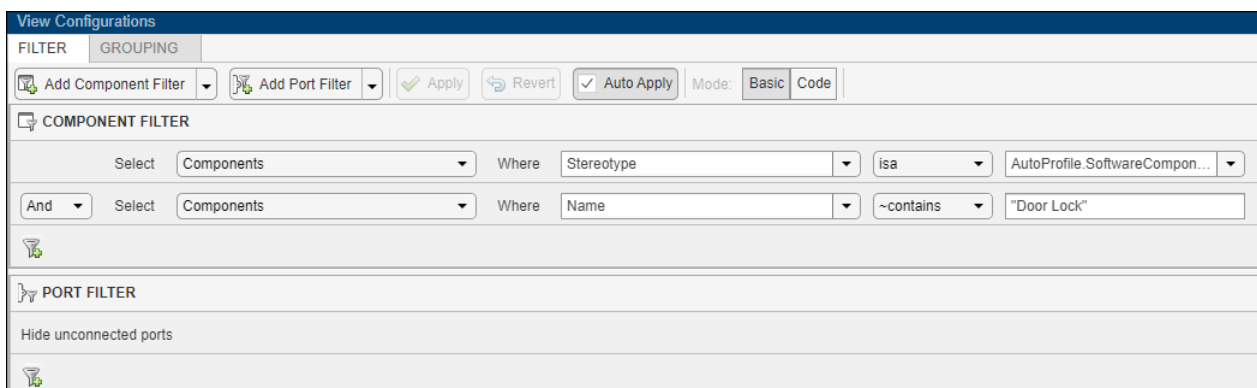
8 Select **Add Component Filter**. From the **Select** list, select **Components**. From the **Where** list, select **Name**. Select **~contains**. In the text box, enter "Door Lock". Select the **Auto Apply** check box so that future changes are applied without selecting **Apply**.

View Configurations			
FILTER		GROUPING	
	Add Component Filter		Add Port Filter
	Apply		Revert
<input checked="" type="checkbox"/>	Auto Apply	Mode:	Basic Code
COMPONENT FILTER			
Select	Components	Where	Stereotype
		isa	AutoProfile.SoftwareCompon...
And	Select	Where	Name
		~contains	"Door Lock"

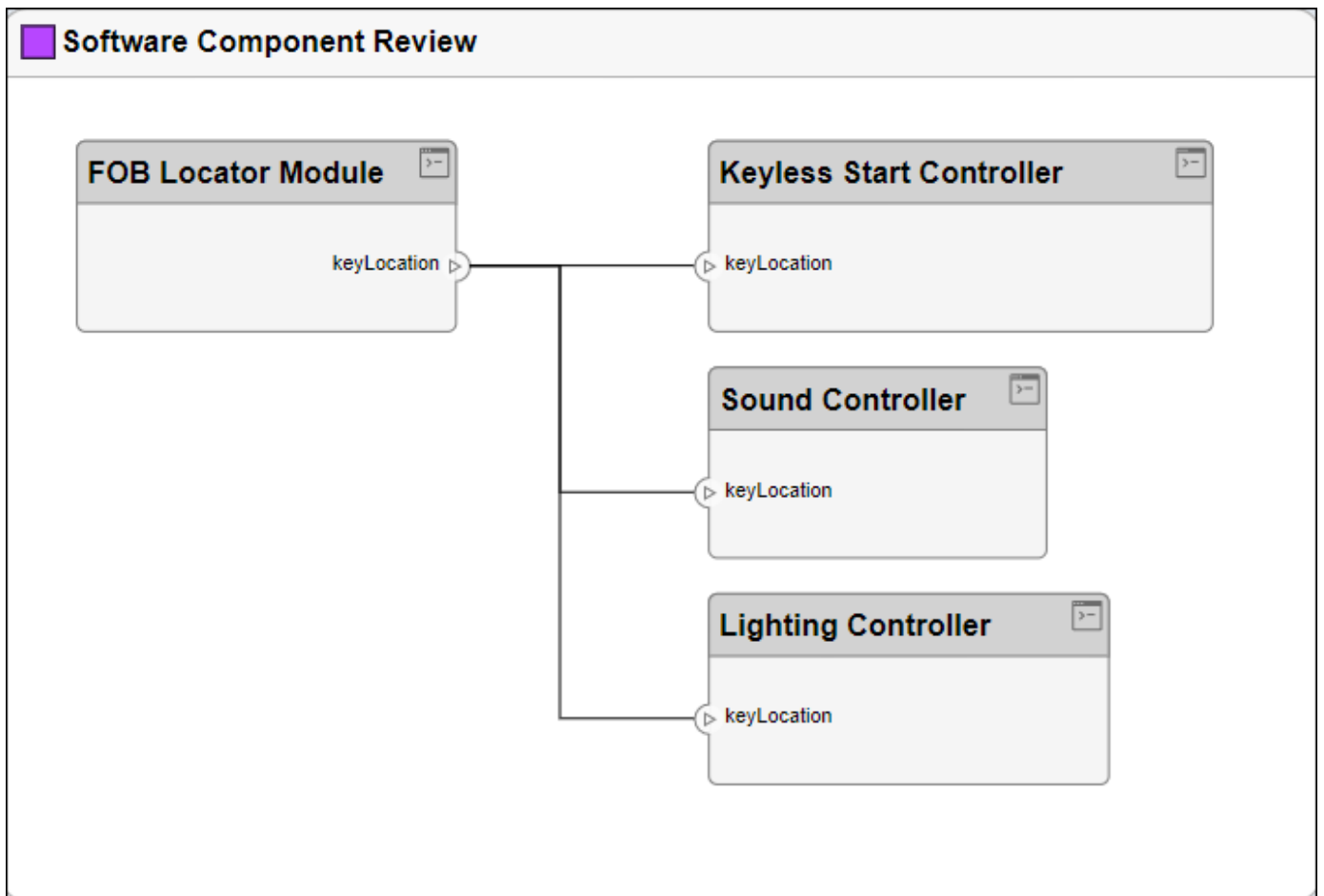
9 An architecture view is created using the additional query in the **Component Filter** box. The view is filtered to select all components not named "Door Lock".




10 From the **Add Port Filter** list, select the option Hide Unconnected Ports.



11 An architecture view is created using the additional query in the **Port Filter** box. The view is filtered to hide unconnected ports.

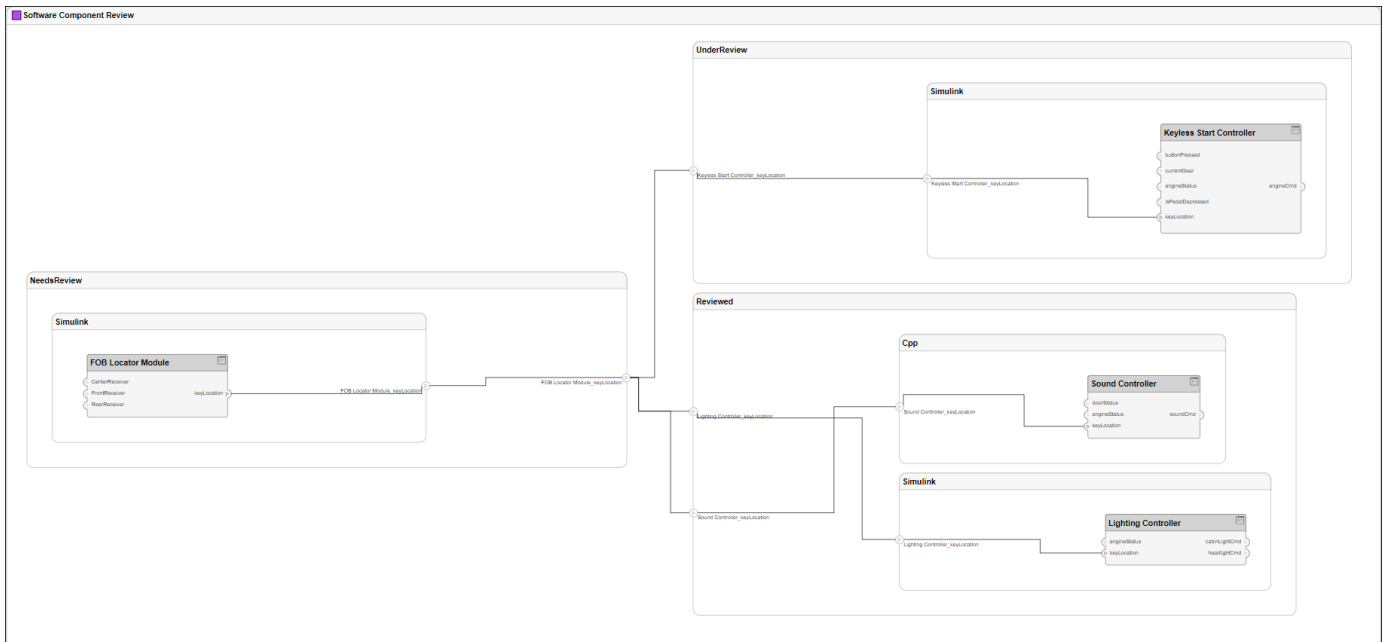


12

Delete the port filter. Pause on the constraint and select the  button.

Add Group By Criteria to Filtered Views

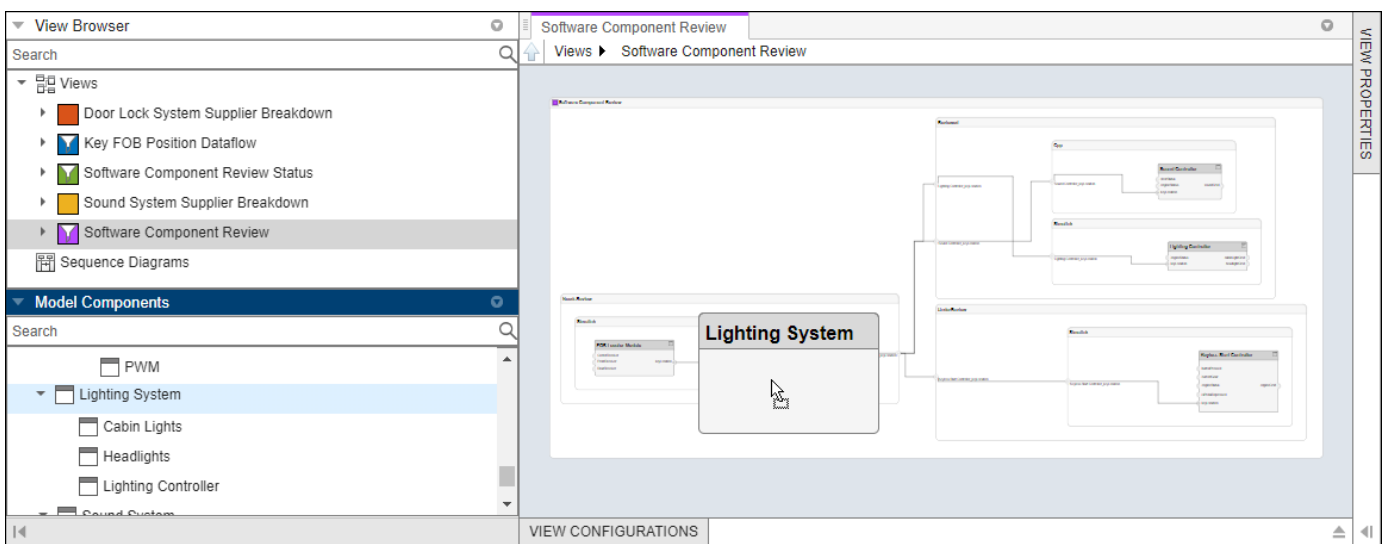
- 1 In the View Configurations pane, select **Grouping**.
- 2 To choose a property enumeration for grouping, click **Add Group By**.
- 3 From the list, select `AutoProfile.BaseComponent.ReviewStatus`.
- 4 Click **Add Group By** again.
- 5 From the list, select `AutoProfile.SoftwareComponent.ImplementationLanguage`.
- 6 Click **Apply**.



Interactively Add and Remove Elements from Views

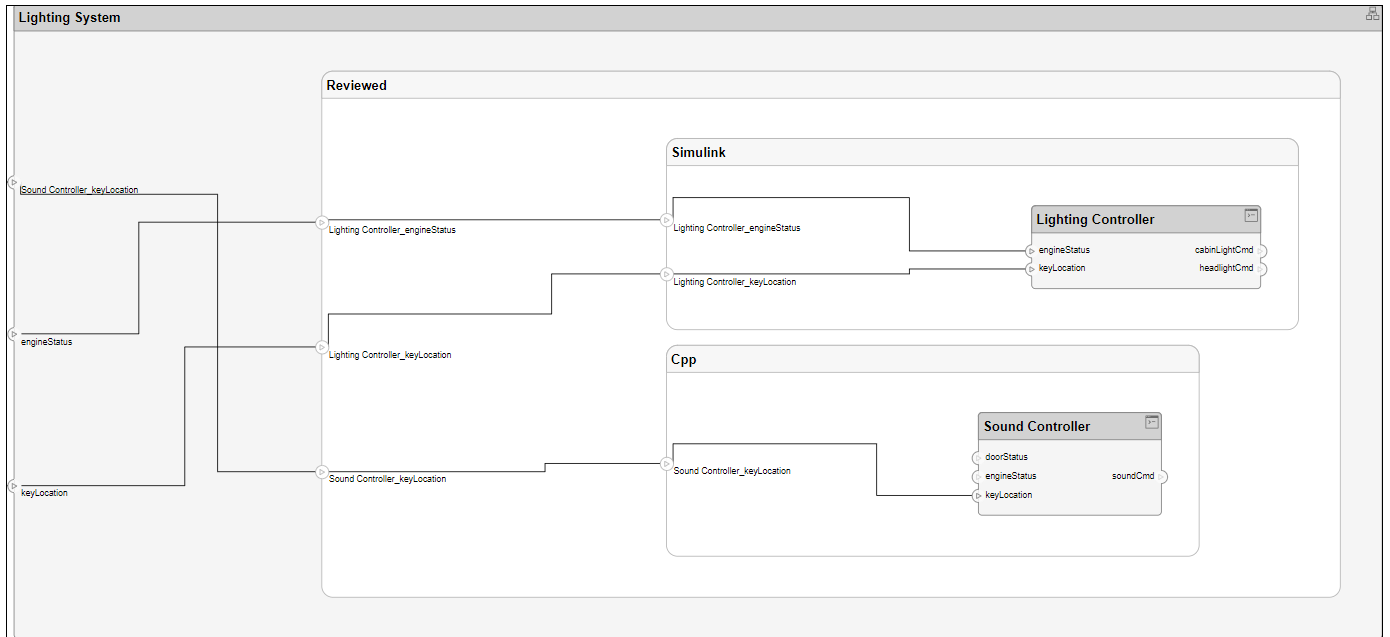
- 1 To add more components to the view, drag and drop components from **Model Components**. Drag and drop the **Lighting System** component to the **Software Component Review** view. Alternatively, click **Add** on the toolstrip. You can also press **Ctrl+I** to add component instantiations to your view 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.



You can press **Delete** to delete components from the view.

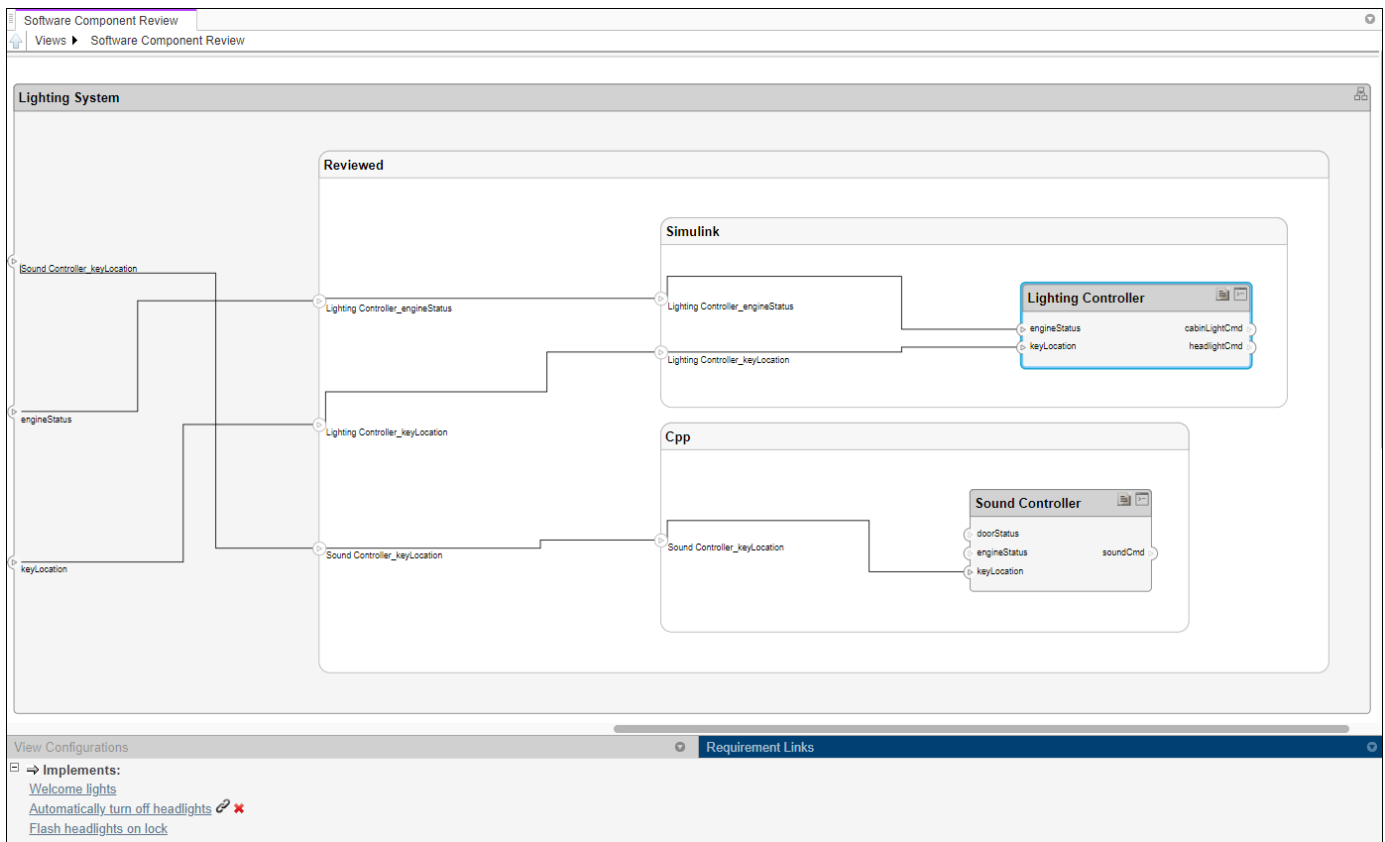
- 2 Observe that the **Lighting System** component has been added to the view.



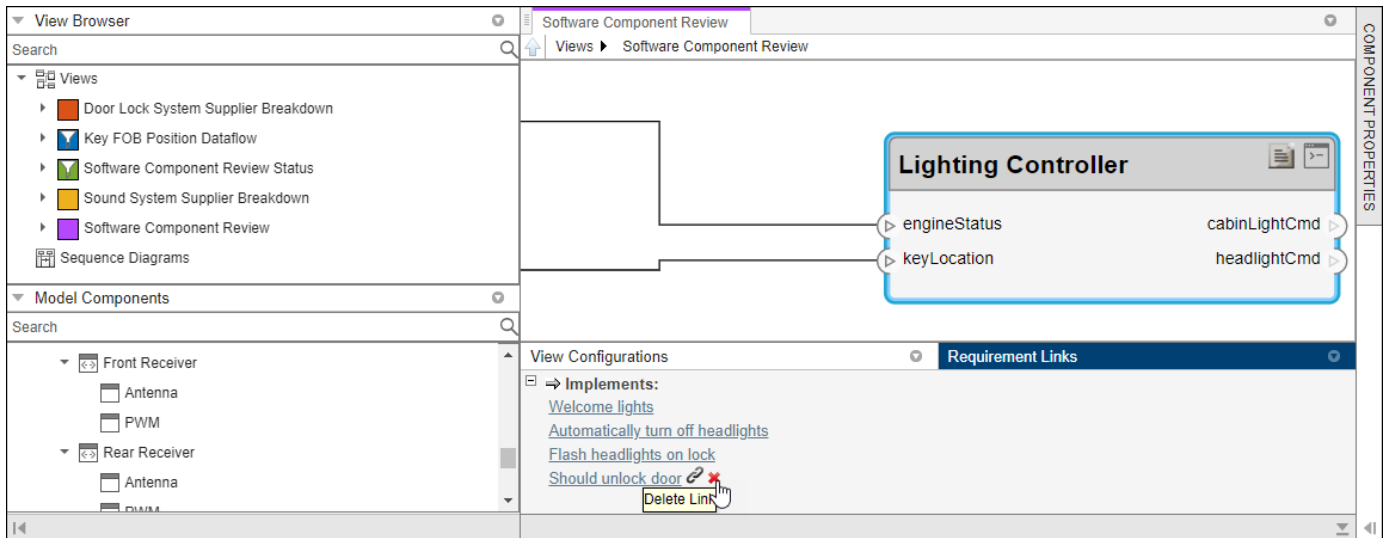
This view is now considered a freeform view.

Add or Remove Requirements Links from Views

- 1 Navigate to **Requirement > Requirements Manager**. A Requirements Toolbox license is required. The **Requirement Links** tab appears at the bottom of the Software Component Review view.
- 2 Select the **Lighting Controller** component and observe the linked requirement **Automatically turn off headlights**.




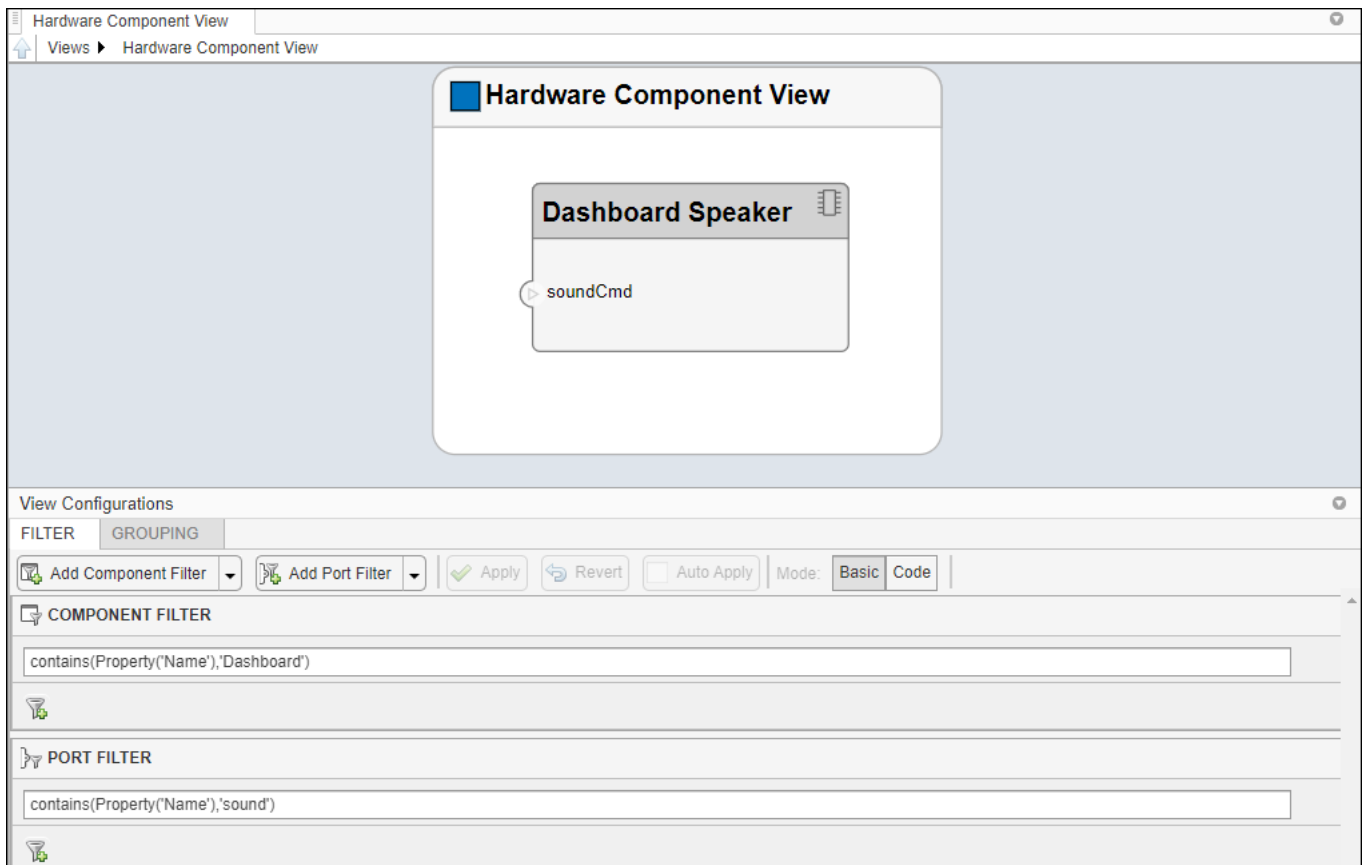
- 3 Select the requirement **Automatically turn off headlights** to open the Requirements Editor to view or modify requirement links.
- 4 In the **Architecture Views Gallery**, navigate to **Requirement > Open Requirements Editor** if the Requirements Editor is not open already.
- 5 Select the **Should unlock door** requirement.
- 6 Return to the **Architecture Views Gallery**. In the **Software Component Review** view, select the **Lighting Controller** component.
- 7 Navigate to **Requirement > Link to selected requirement**. The new requirement **Should unlock door** is added.



- 8 To remove a requirement link, select **X** and confirm deletion.

Add Custom Clauses to Component Filters and Port Filters

- 1 Select **New > View** to create a new view.
- 2 In **View Properties** on the right pane, in the **Name** box, enter a name for this view, for example, Hardware Component View. Choose a **Color** and enter a **Description**, if necessary.
- 3 In the bottom pane, under **View Configurations > Filter**, select from the list **Add Component Filter > Add Custom Component Filter** to enter a constraint by which to filter. In the box, enter `contains(Property('Name'), 'Dashboard')`.
- 4 In the bottom pane, under **View Configurations > Filter**, select from the list **Add Port Filter > Add Custom Port Filter** to enter a constraint by which to filter. In the box, enter `contains(Property('Name'), 'sound')`.
- 5 Select **Apply** .



The view is filtered using the constraints in the custom filters. For more information on structuring constraints, see `systemcomposer.query.Constraint`.

See Also

More About

- “Create Architectural Views Programmatically” on page 9-15
- “Display Component Hierarchy and Architecture Hierarchy Using Views” on page 9-21
- “Create Spotlight Views” on page 9-2
- “Modeling System Architecture of Keyless Entry System” on page 9-25

Create Architectural Views Programmatically

You can create an architecture view programmatically. This topic presents two examples of creating architecture views programmatically using a keyless entry system architecture using element groups.

An element group is a grouping of components in a view.

Use element groups to programmatically populate a view.

For more information on the keyless entry system architecture, see “Modeling System Architecture of Keyless Entry System” on page 9-25.

The third example is about 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.

Create Architecture Views in System Composer with Keyless Entry System

Use a keyless entry system to programmatically create architecture views.

1. Import the package with queries.

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

2. Open the Simulink® project file for the Keyless Entry System.

```
scKeylessEntrySystem
```

3. Load the example model into System Composer™.

```
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Example 1: Hardware Component Review Status View

Create a filtered view that selects all hardware components in the architecture model and groups them using the ReviewStatus property.

1. Construct a query to select all hardware components.

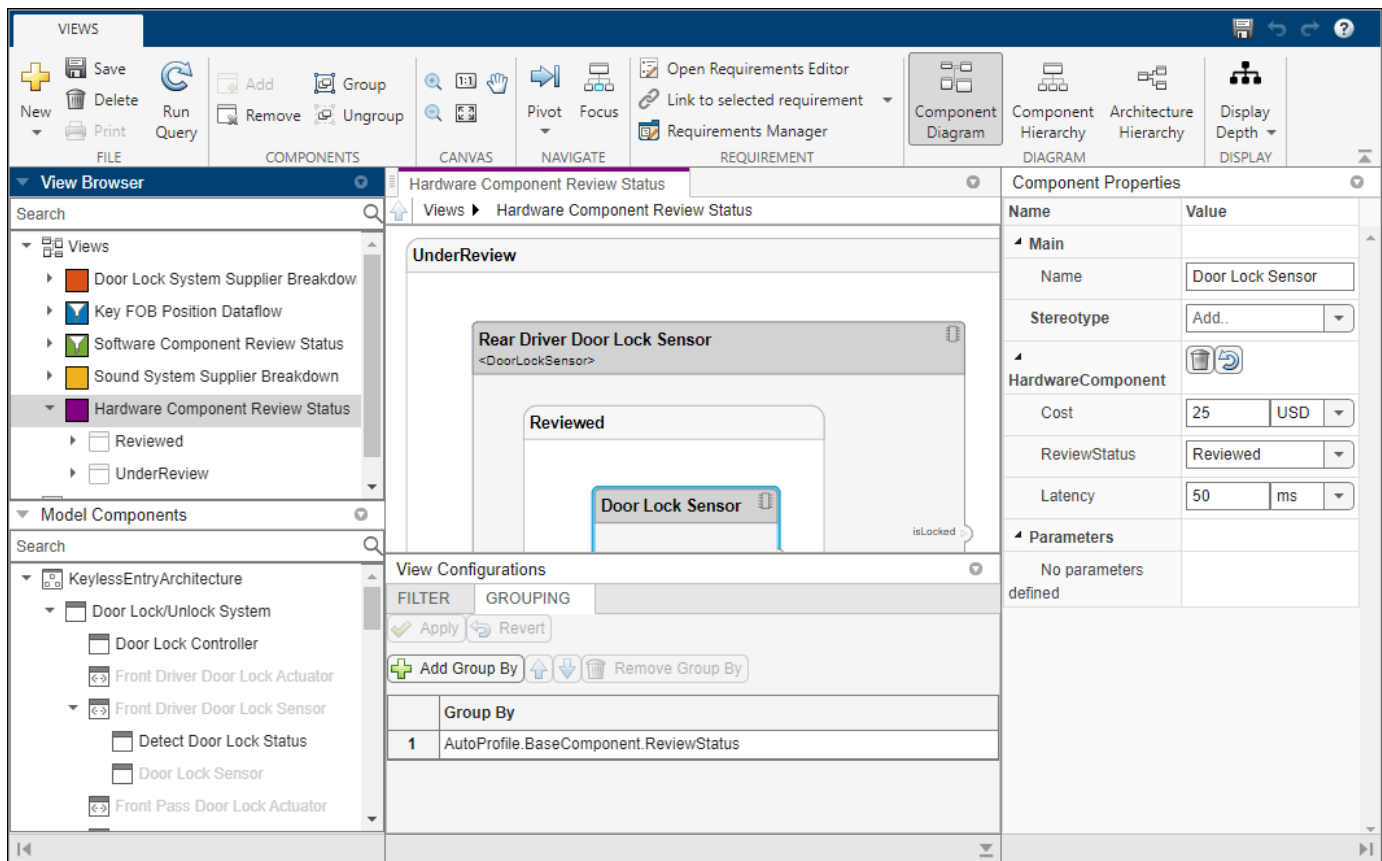
```
hwCompQuery = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent"));
```

2. Use the query to create a view.

```
model.createView("Hardware Component Review Status", ...
    Select=hwCompQuery, ...
    GroupBy={'AutoProfile.BaseComponent.ReviewStatus'}, ...
    IncludeReferenceModels=true, ...
    Color="purple");
```

3. To open the Architecture Views Gallery the **Views** section, click **Architecture Views**.

```
model.openViews
```



Example 2: FOB Locator System Supplier View

Create a freeform view that manually pulls the components from the FOB Locator System and groups them using existing and new view components for the suppliers. In this example, you will use *element groups*, groupings of components in a view, to programmatically populate a view.

1. Create a view architecture.

```
fobSupplierView = model.createView("FOB Locator System Supplier Breakdown", ...
    Color="lightblue");
```

2. Add a subgroup called Supplier D. Add the FOB Locator Module to the view element subgroup.

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

3. Create a new subgroup for Supplier A.

```
supplierA = fobSupplierView.Root.createSubGroup("Supplier A");
```

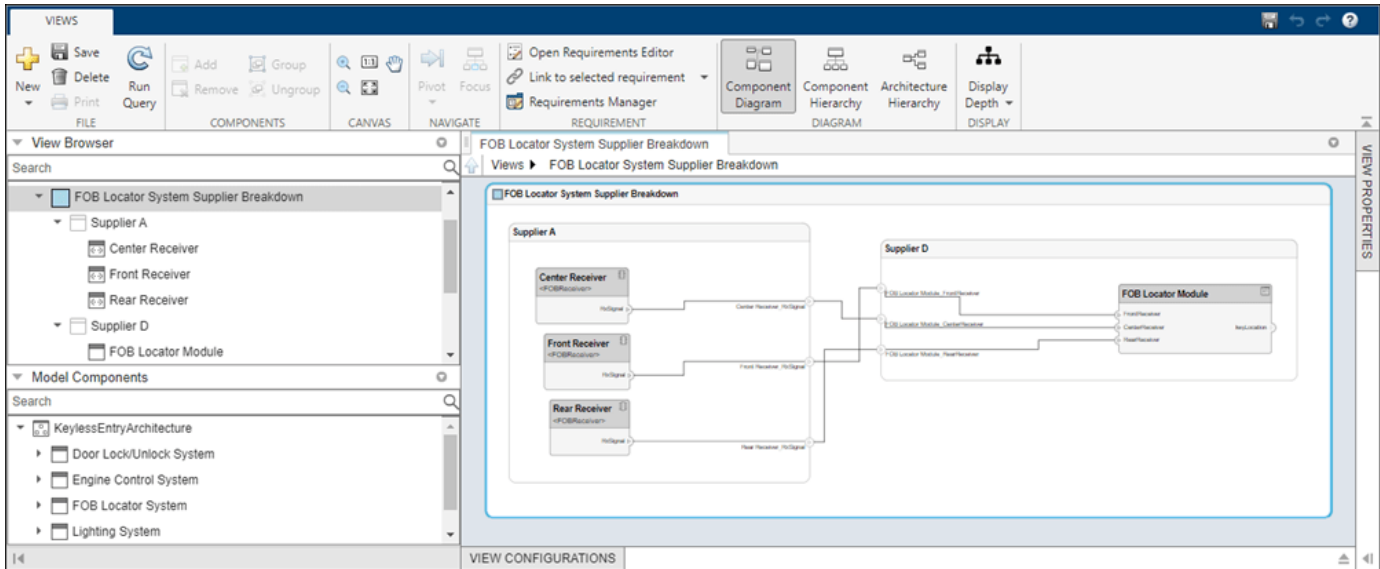
4. Add each of the FOB Receivers to view element subgroup.

```
FOBLocatorSystem = model.lookup("Path", "KeylessEntryArchitecture/FOB Locator System");
```

Find all the components which contain the name "Receiver".

```
receiverCompPaths = model.find(...
    contains(Property("Name"), "Receiver"), ...
    FOBLocatorSystem.Architecture);

supplierA.addElement(receiverCompPaths)
```



5. Save the model.

```
model.save
```

Find Elements in Model Using Queries

Find components in a System Composer model using queries.

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

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

Open the model.

```
scKeylessEntrySystem
model = systemcomposer.loadModel("KeylessEntryArchitecture");
```

Find all the software components in the system.

```
con1 = HasStereotype(Property("Name") == "SoftwareComponent");
[compPaths, compObjs] = model.find(con1)
```

```
compPaths = 5x1 cell
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
    {'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module' }
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller' }
    {'KeylessEntryArchitecture/Lighting System/Lighting Controller' }
    {'KeylessEntryArchitecture/Sound System/Sound Controller' }
    }
```

```
compObjs=1x5 object
    1x5 Component array with properties:
```

```

IsAdapterComponent
Architecture
ReferenceName
Name
Parent
Ports
OwnedPorts
OwnedArchitecture
Position
Model
SimulinkHandle
SimulinkModelHandle
UUID
ExternalUID

```

Include reference models in the search.

```
softwareComps = model.find(con1, IncludeReferenceModels=true)
```

```
softwareComps = 9x1 cell
```

```

{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller'
{'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module'
{'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller'
{'KeylessEntryArchitecture/Lighting System/Lighting Controller'
{'KeylessEntryArchitecture/Sound System/Sound Controller'
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Sensor/Detect Door L
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Sensor/Detect Door
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Sensor/Detect Door
{'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 = model.find(con2)
```

```
baseComps = 18x1 cell
```

```

{'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
{'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module' }
{'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller' }
{'KeylessEntryArchitecture/Lighting System/Lighting Controller' }
{'KeylessEntryArchitecture/Sound System/Sound Controller' }
{'KeylessEntryArchitecture/Engine Control System/Start//Stop Button' }
{'KeylessEntryArchitecture/Sound System/Dashboard Speaker' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Sensor' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Sensor' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Sensor' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Sensor' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Driver Door Lock Actuator' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Front Pass Door Lock Actuator' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Driver Door Lock Actuator' }
{'KeylessEntryArchitecture/Door Lock//Unlock System/Rear Pass Door Lock Actuator' }
{'KeylessEntryArchitecture/FOB Locator System/Center Receiver' }
{'KeylessEntryArchitecture/FOB Locator System/Front Receiver' }
{'KeylessEntryArchitecture/FOB Locator System/Rear Receiver' }

```

Find all components using the interface KeyFOBPosition.

```
con3 = HasPort(HasInterface(Property("Name") == "KeyFOBPosition"));
con3_a = HasPort(Property("InterfaceName") == "KeyFOBPosition");
keyFOBPosComps = model.find(con3)

keyFOBPosComps = 10x1 cell
    {'KeylessEntryArchitecture/Door Lock//Unlock System' }
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
    {'KeylessEntryArchitecture/Engine Control System' }
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller' }
    {'KeylessEntryArchitecture/FOB Locator System' }
    {'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module' }
    {'KeylessEntryArchitecture/Lighting System' }
    {'KeylessEntryArchitecture/Lighting System/Lighting Controller' }
    {'KeylessEntryArchitecture/Sound System' }
    {'KeylessEntryArchitecture/Sound System/Sound Controller' }
```

Find all components whose WCET is less than or equal to 5 ms.

```
con4 = PropertyValue("AutoProfile.SoftwareComponent.WCET") <= 5;
model.find(con4)

ans = 1x1 cell array
    {'KeylessEntryArchitecture/Sound System/Sound Controller'}
```

You can specify units for automatic unit conversion.

```
con5 = PropertyValue("AutoProfile.SoftwareComponent.WCET") <= Value(5, 'ms');
query1Comps = model.find(con5)

query1Comps = 3x1 cell
    {'KeylessEntryArchitecture/Sound System/Sound Controller' }
    {'KeylessEntryArchitecture/FOB Locator System/FOB Locator Module' }
    {'KeylessEntryArchitecture/Lighting System/Lighting Controller' }
```

Find all components whose WCET is greater than 1 ms or that have a cost greater than 10 USD.

```
con6 = PropertyValue("AutoProfile.SoftwareComponent.WCET") > Value(1, 'ms') | PropertyValue("AutoProfile.SoftwareComponent.Cost") > 10;
query2Comps = model.find(con6)

query2Comps = 2x1 cell
    {'KeylessEntryArchitecture/Door Lock//Unlock System/Door Lock Controller' }
    {'KeylessEntryArchitecture/Engine Control System/Keyless Start Controller' }
```

Close the model.

```
model.close
```

See Also

[find](#) | [lookup](#) | [systemcomposer.query.Constraint](#) | [createView](#) | [getView](#) | [openViews](#) | [deleteView](#) | [systemcomposer.view.View](#) | [systemcomposer.view.ElementGroup](#)

More About

- “Create Architecture Views Interactively” on page 9-5
- “Display Component Hierarchy and Architecture Hierarchy Using Views” on page 9-21
- “Create Spotlight Views” on page 9-2
- “Modeling System Architecture of Keyless Entry System” on page 9-25

Display Component Hierarchy and Architecture Hierarchy Using Views

This example shows how to use hierarchy views in the **Architecture Views Gallery** to use hierarchy views to visualize hierarchical relationships.

You can visualize a hierarchy diagram as a view with components, ports, reference types, component stereotypes, and stereotype properties.

There are two types of hierarchy diagrams:

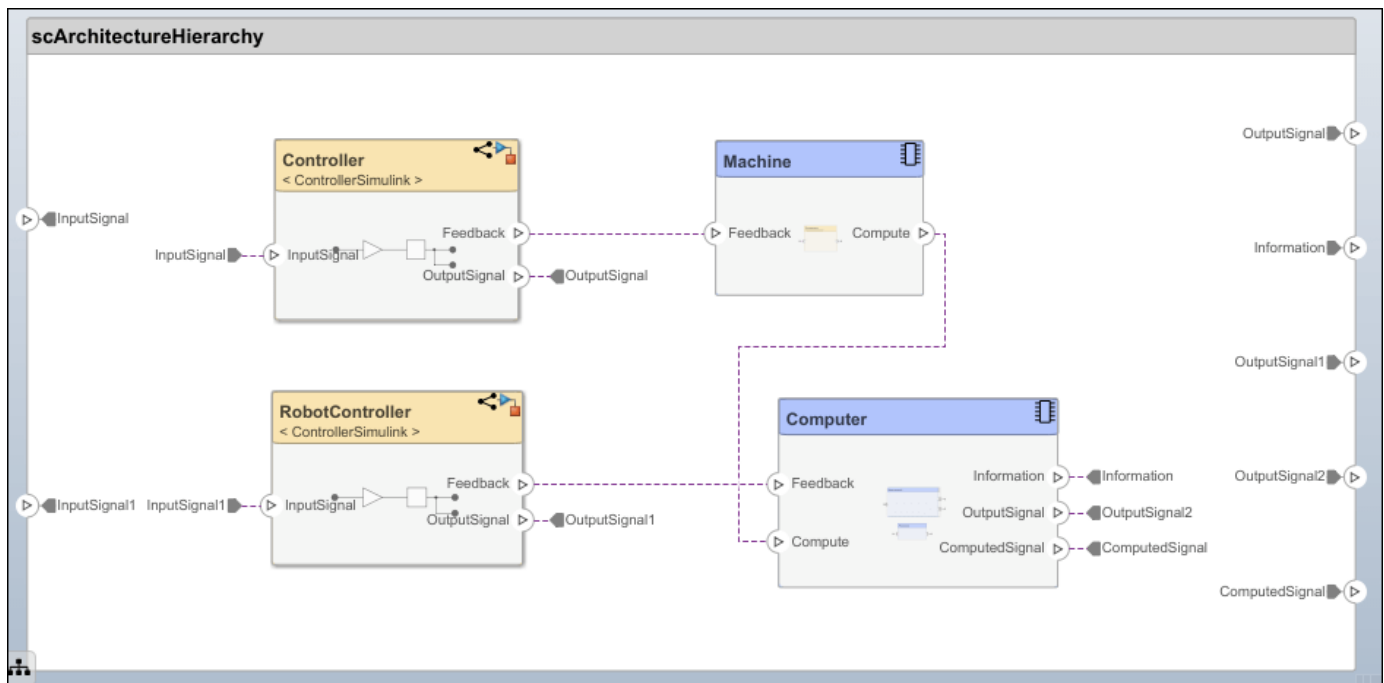
- *Component hierarchy diagrams* display components in tree form with parents above children. In a component hierarchy view, each referenced model is represented as many times as it is used.
- *Architecture hierarchy diagrams* display unique component architecture types and their relationships using composition connections. In an architecture hierarchy view, each referenced model is represented only once.

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

This example uses an architecture model representing data flow within a robotic system. Open this model to follow the steps in the tutorial.

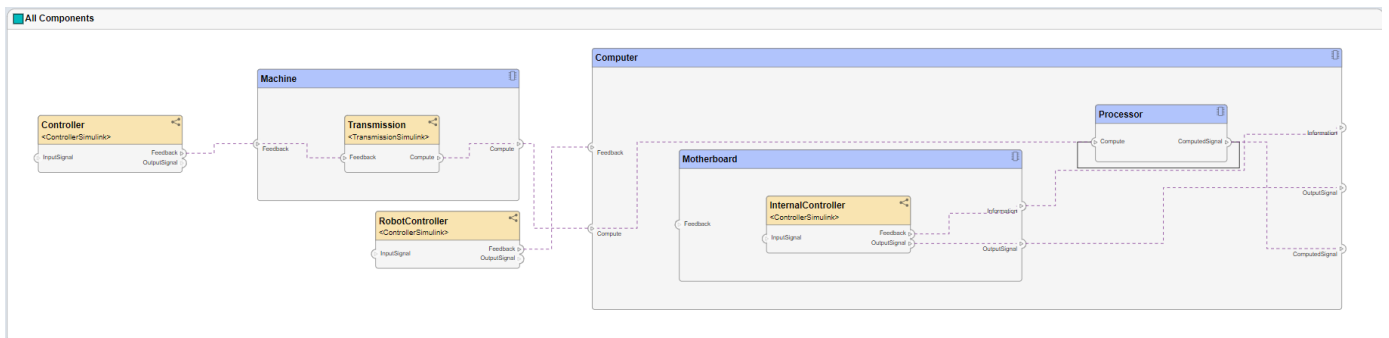
Robot Computer Systems Architecture

Use a robot computer system with controllers that simulate transmission of data to explore hierarchy diagrams in the Architecture Views Gallery.



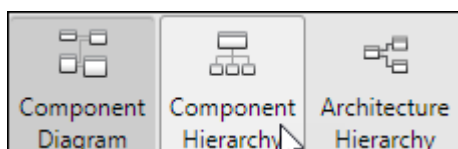
Switch Between Component Diagram View and Hierarchy Views

- 1 To open the **Architecture Views Gallery**, navigate to **Modeling > Architecture Views**.
- 2 From the **View Browser**, select the **All Components** view.
- 3 Observe the component diagram view that corresponds to the all the components in the architecture model.

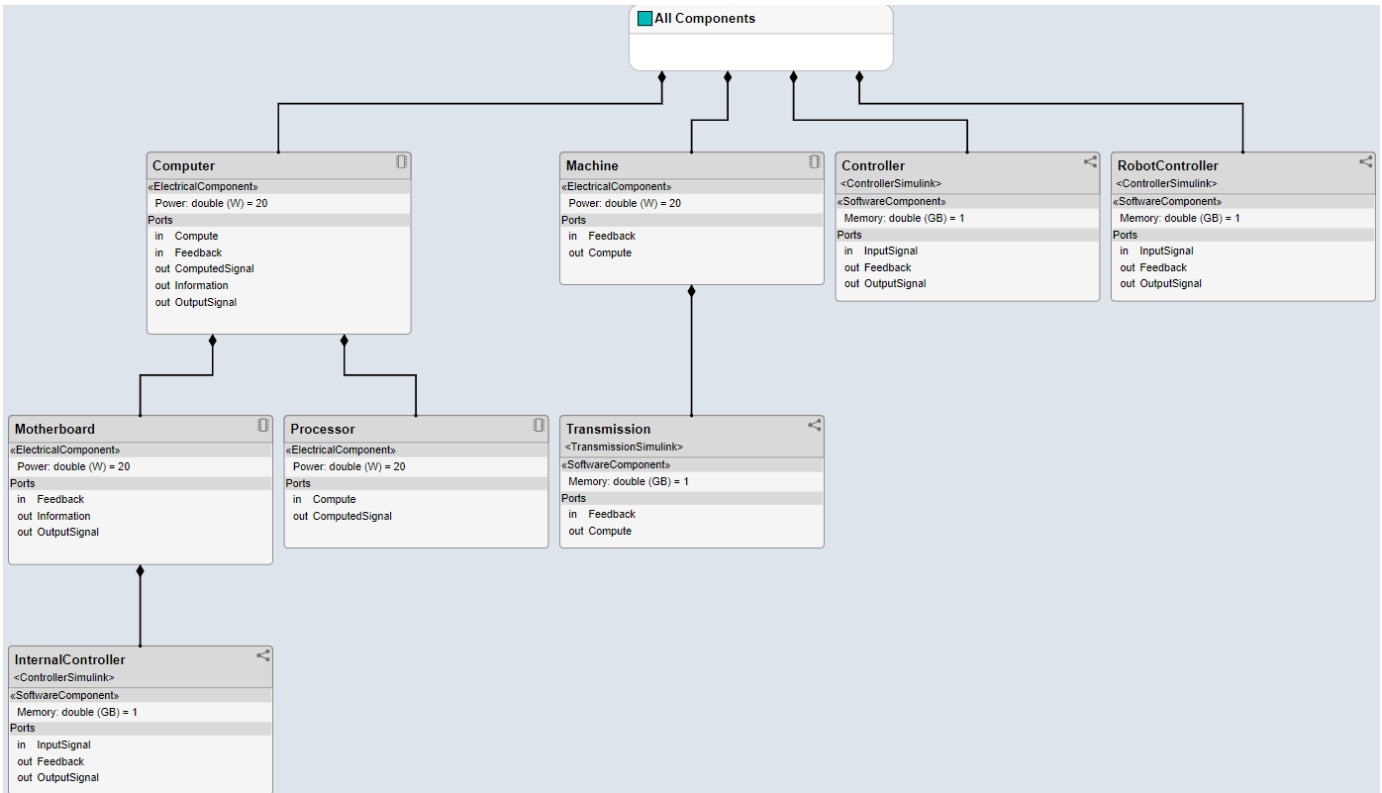


The component diagram represents a view with components, ports, and connectors based on how the model is structured.

- 4 Click **Diagram > Component Hierarchy**.

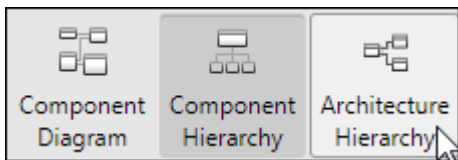


- Observe the component hierarchy view that corresponds to the same set of components.

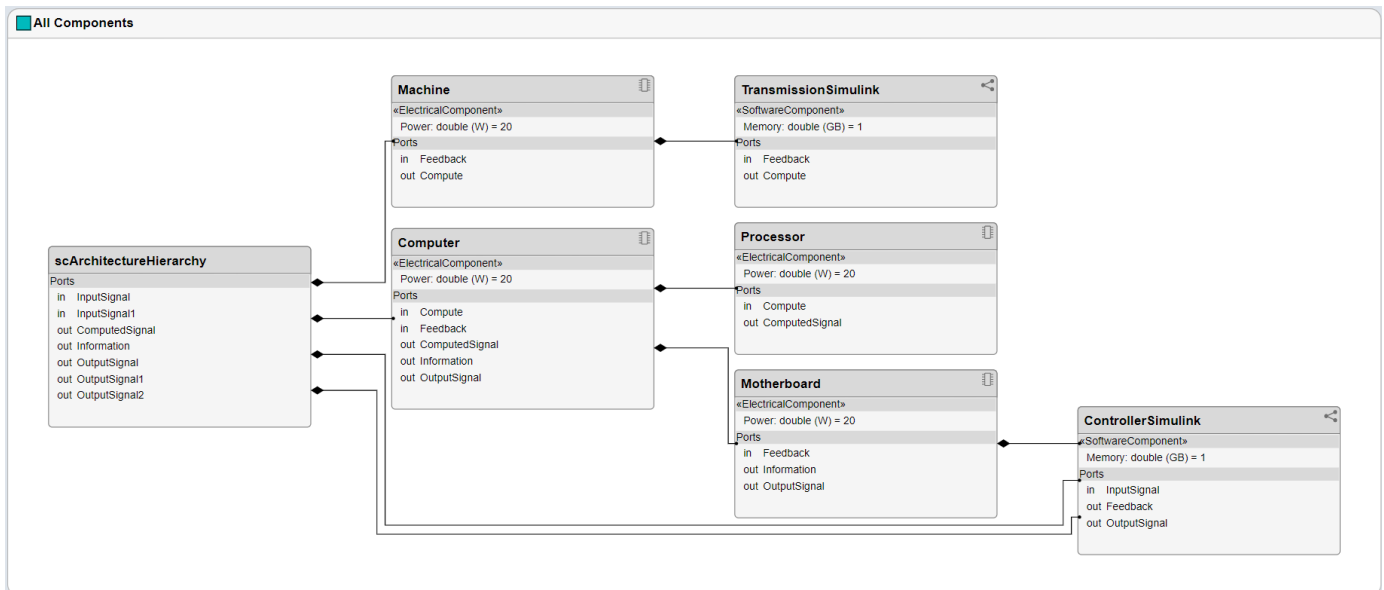


The component hierarchy diagram shows a single root, which is the view specification itself. The root corresponds to the name of the view shown in the component diagram. The connections in the component hierarchy diagram originate from the child components and end with a diamond symbol at each parent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

- Click **Diagram > Architecture Hierarchy**.



- Observe the architecture hierarchy view that corresponds to the same set of components.



The architecture hierarchy diagram starts with the root architecture. The root corresponds to the boundary of the system. A box in an architecture hierarchy diagram represents a referenced model and appears only once even if it is referenced multiple times in the same model. For example, **ControllerSimulink**, a referenced model that appears on three components, has three connections to its parent architectures. The connectivity of the boxes represents the relationship between **ControllerSimulink** and its parents.

See Also

More About

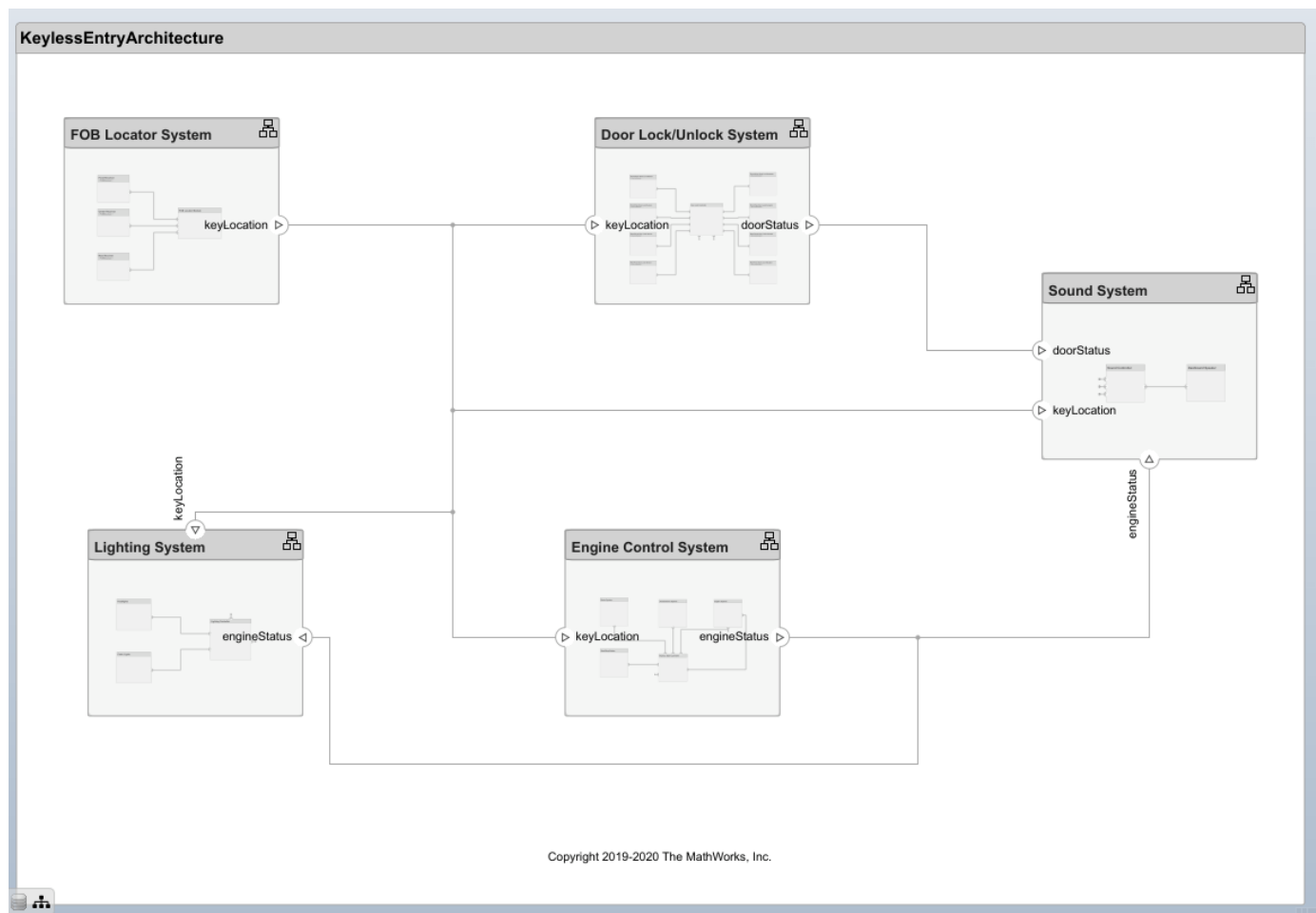
- “Create Architectural Views Programmatically” on page 9-15
- “Create Architecture Views Interactively” on page 9-5
- “Create Spotlight Views” on page 9-2
- “Modeling System Architecture of Keyless Entry System” on page 9-25
- “Class Diagram View of Software Architectures” on page 8-20

Modeling System Architecture of Keyless Entry System

This example shows how to set up the architecture for a keyless entry system for a vehicle in System Composer™. You also learn how to create different architecture views for different stakeholder concerns. This example follows a model-based systems engineering (MBSE) workflow:

- 1 Define Stakeholder Requirements
- 2 Define Logical Architecture Model
- 3 Define Stereotypes to Classify Components
- 4 Define Port Interfaces to Describe Data Flow
- 5 Create Views to Present to Stakeholders

sckeylessentrysystem



Define Stakeholder Requirements

In MBSE design, functional requirements represent high-level stakeholder requirements based on needs and concerns for the design to address. Run this command to open the Requirements Editor (Requirements Toolbox) with the functional requirements. A Requirements Toolbox™ license is required to inspect requirements in a System Composer architecture model.

```
slreq.load('FunctionalRequirements');
slreq.editor
```

Index	ID	Summary
FunctionalRequirements		
1	#1	Should unlock door
2	#2	Should lock door
3	#3	Welcome lights
4	#4	Automatically turn off headlights
5	#5	Flash headlights on lock
6	#6	Emit sound when car unlocked
7	#7	Emit sound when car locked
8	#8	Emit sound when car is on but not key detected
9	#9	Engine start with push button
10	#10	Engine stop with push button

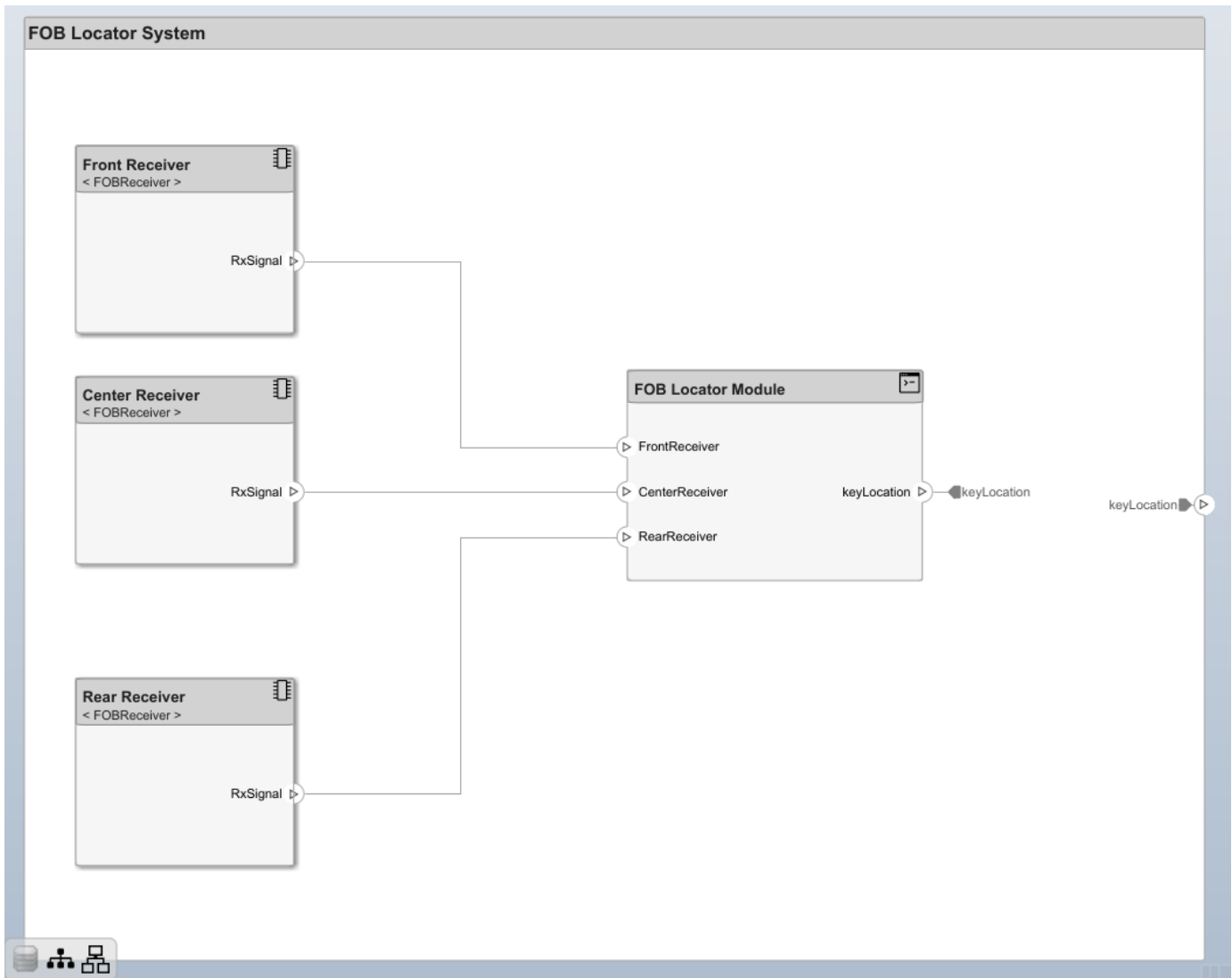
These stakeholder requirements specify that the architecture model must include a door lock and unlock system, a lighting control system, a sound system, and an engine control system. These components should meet requirements after passing quality checks. For more information, see “Manage Requirements” on page 2-8.

Define Logical Architecture Model

The logical architecture of a keyless entry system includes sensors, a mechanical door lock system, a lighting system, a sound system, and an engine control system. These components interact based on the information passed through their ports by connections. Each top-level component can be decomposed into its subcomponents to represent an architectural hierarchy.

Decompose the FOB Locator System

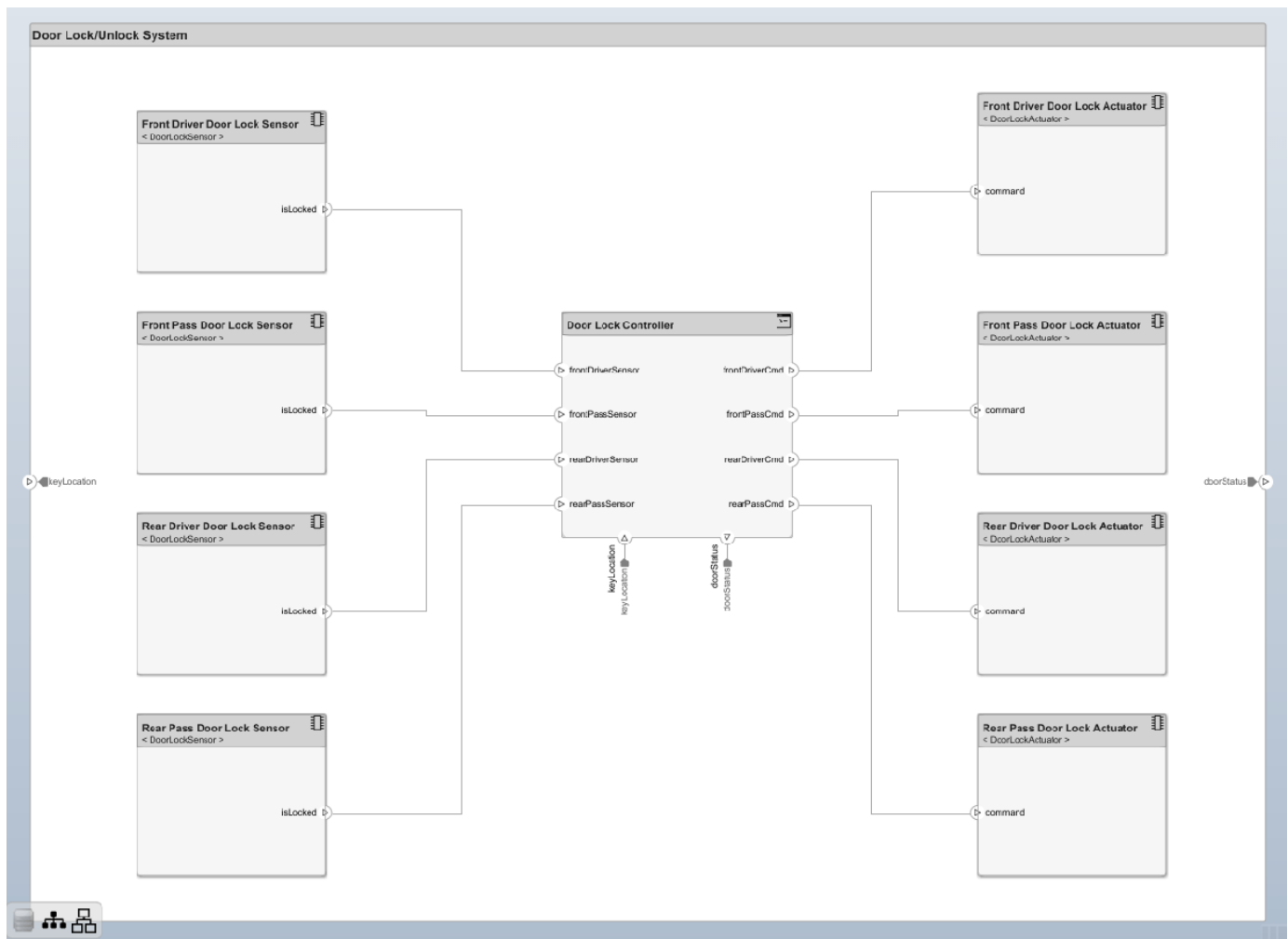
The FOB Locator System component includes the system the vehicle uses to receive a wireless signal and isolate the location of the key to lock or unlock doors. This action is the first step in implementing a keyless entry system.



The referenced architecture `FOB Receiver` is reused to type the three receivers on the vehicle on the front, center, and rear. Each receiver sends a signal, `RxSignal`, to the `FOB Locator Module` component that determines the key location, `keyLocation`, and transmits the key location to all the other systems represented by components.

Decompose the Door Lock and Unlock System

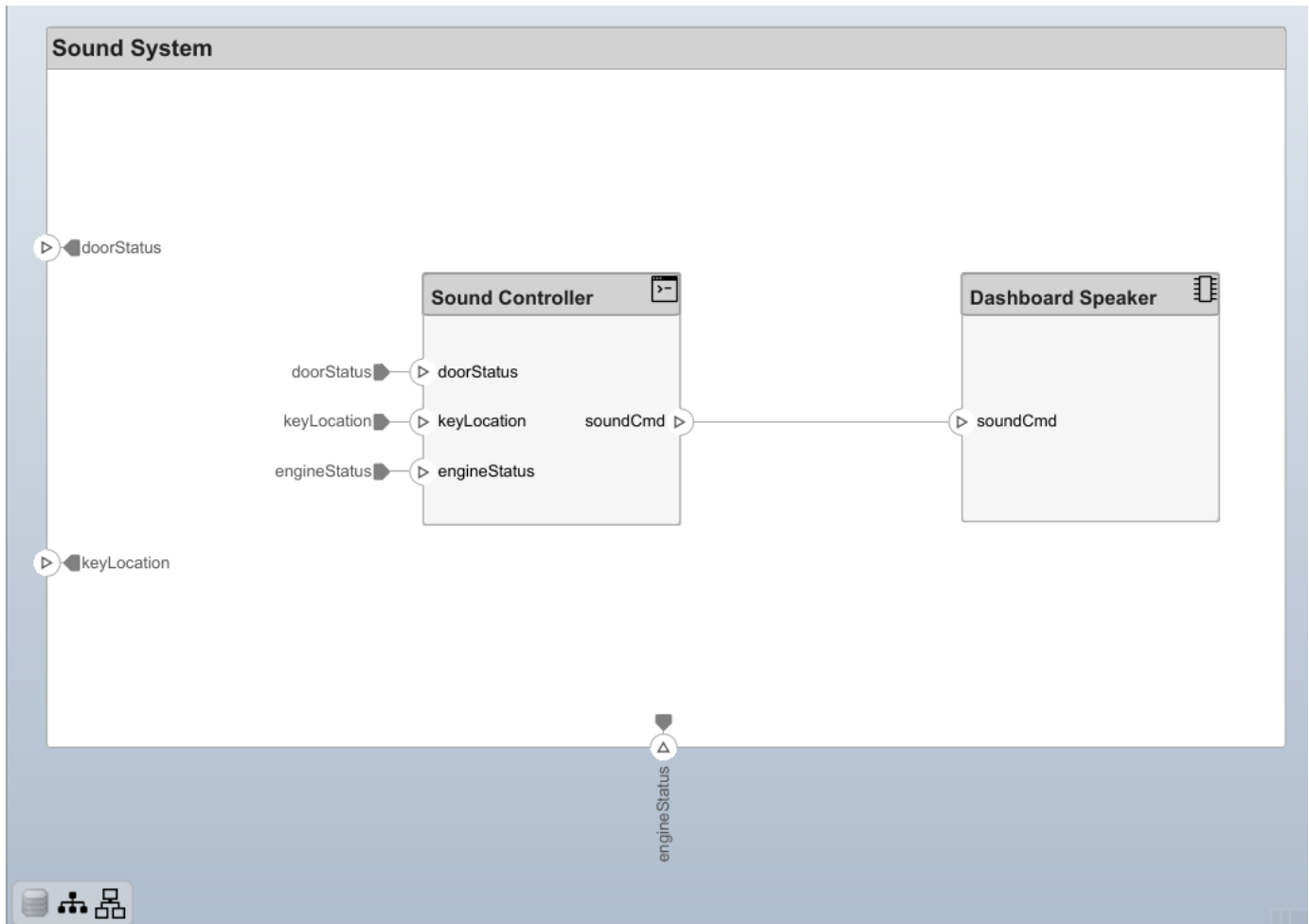
After the key location is identified, the automated door lock and unlock system activates. The door lock and unlock system has both hardware and software components.



All four doors for the vehicle have sensors that detect whether the door is locked. The DoorLockSensor referenced architecture types these components. According to the information passed through the FOB, the Door Lock Controller component activates the actuators for the four doors to optionally implement the door locks. The DoorLockActuator referenced architecture types the actuators. The door lock status doorStatus is transmitted through the boundary from the Door Lock Controller.

Decompose the Sound System

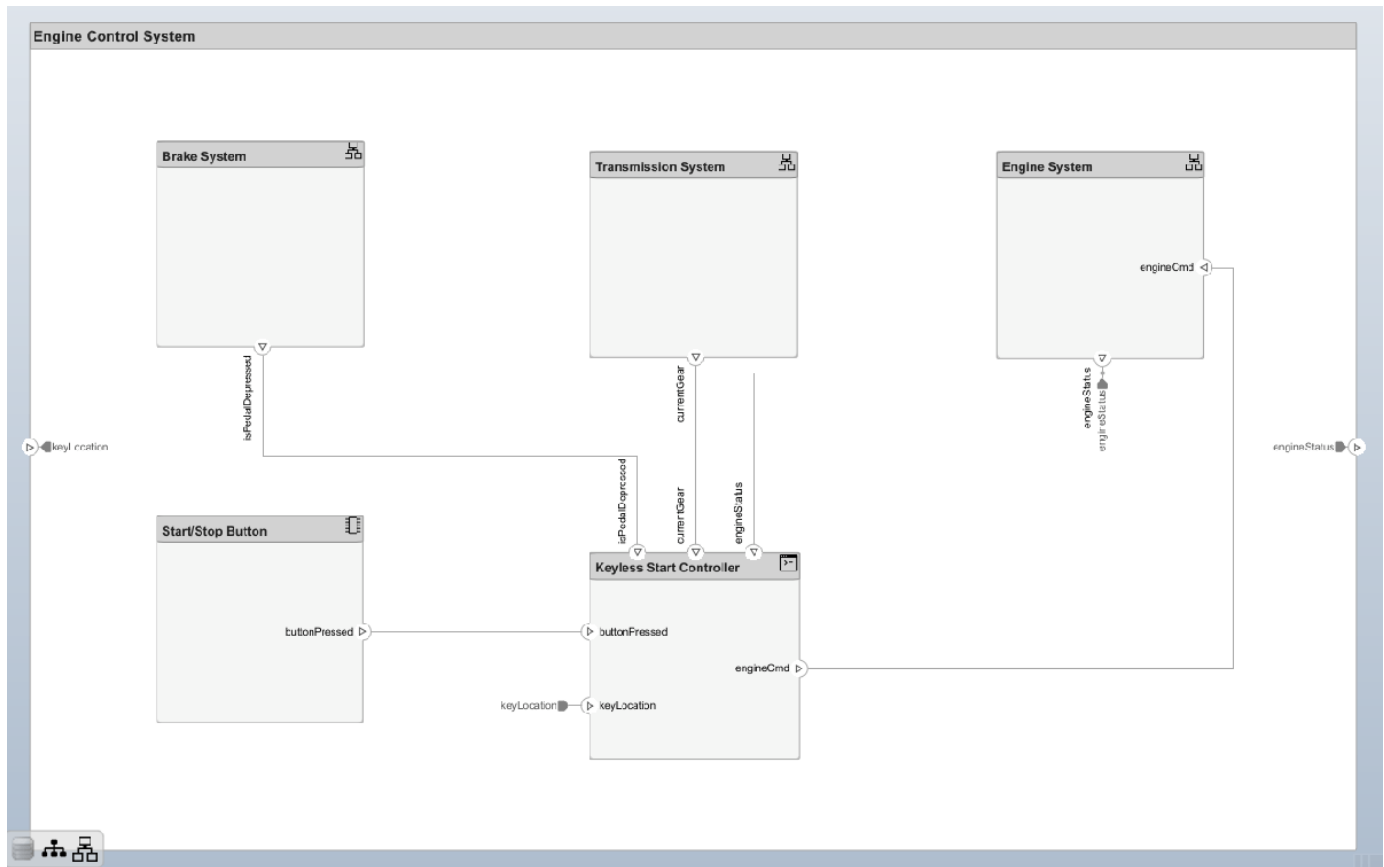
The Sound System component controls emitting sounds to indicate key location, door status, and engine status.



The software for the Sound Controller component sends a command to the Dashboard Speaker component to process and make the sounds.

Decompose the Engine Control System

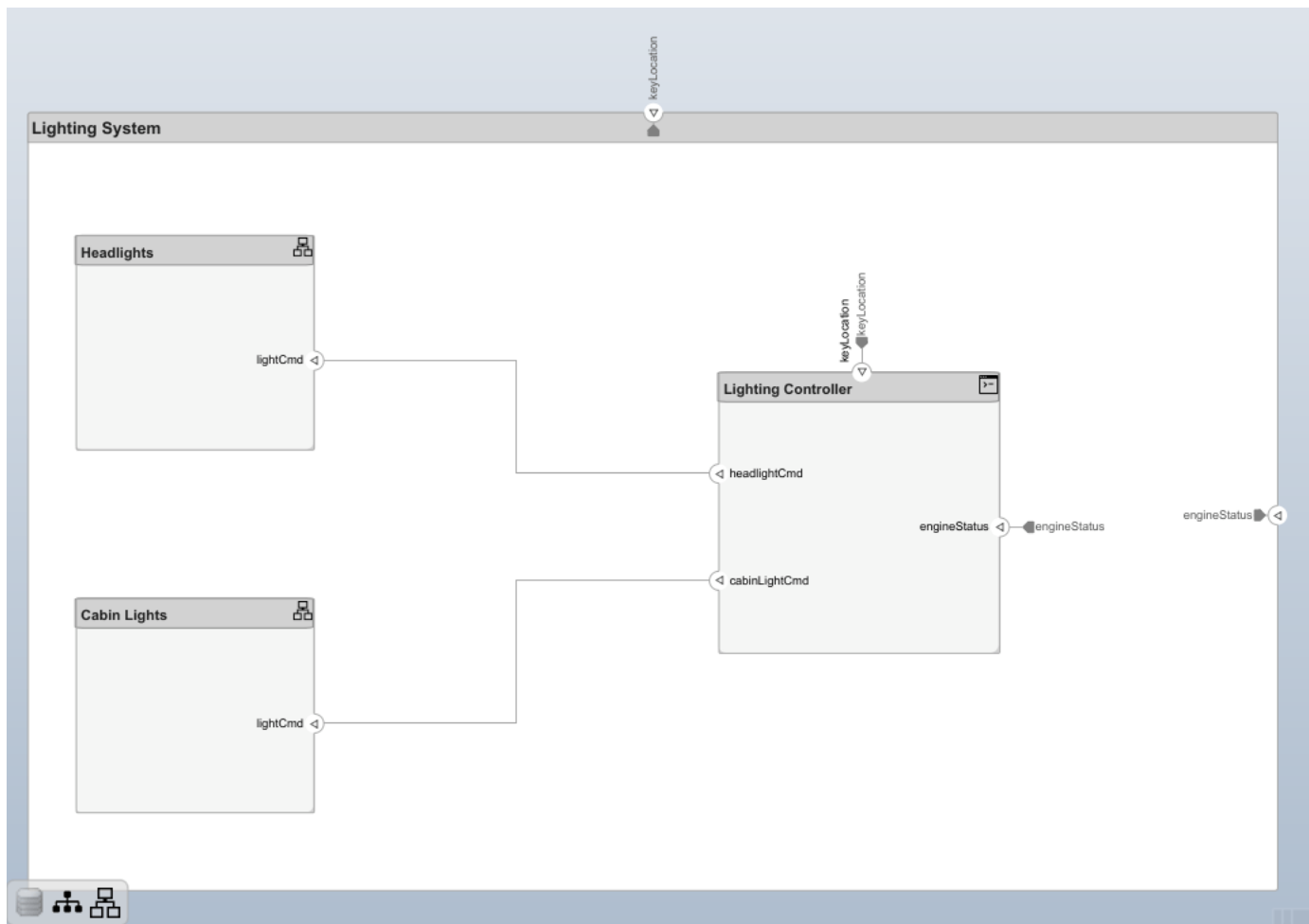
The engine control system contains a system to control the brakes, transmission, and engine. The keyless entry system activates a keyless start controller that starts or stops the vehicle.



The keyLocation signal from the FOB Locator System enters the Keyless Start Controller component, which receives feedback from the Brake System, Transmission System, and Engine System components. Next, a signal is sent to the Engine System to start or stop the vehicle according to the buttonPressed signal from the Start/Stop Button component.

Decompose the Lighting System

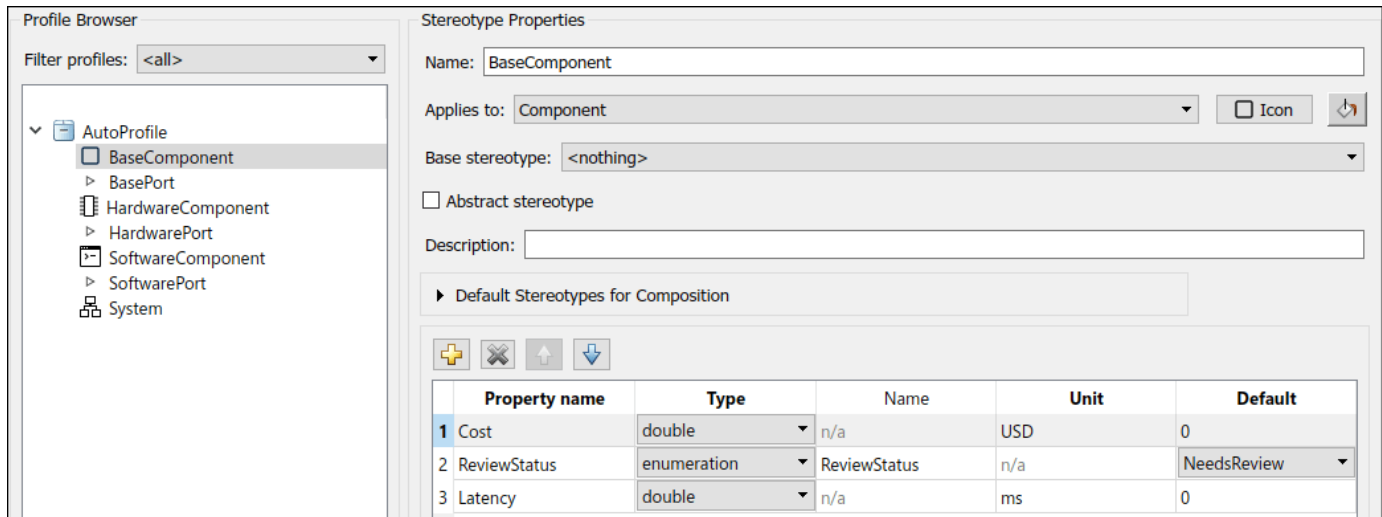
The keyless entry system sends commands from a lighting controller to activate the headlights and cabin lights.



Engine status and key location information sent to the Lighting Controller component might activate the different lights represented by the Headlights and Cabin Lights components.

Define Stereotypes to Classify Components

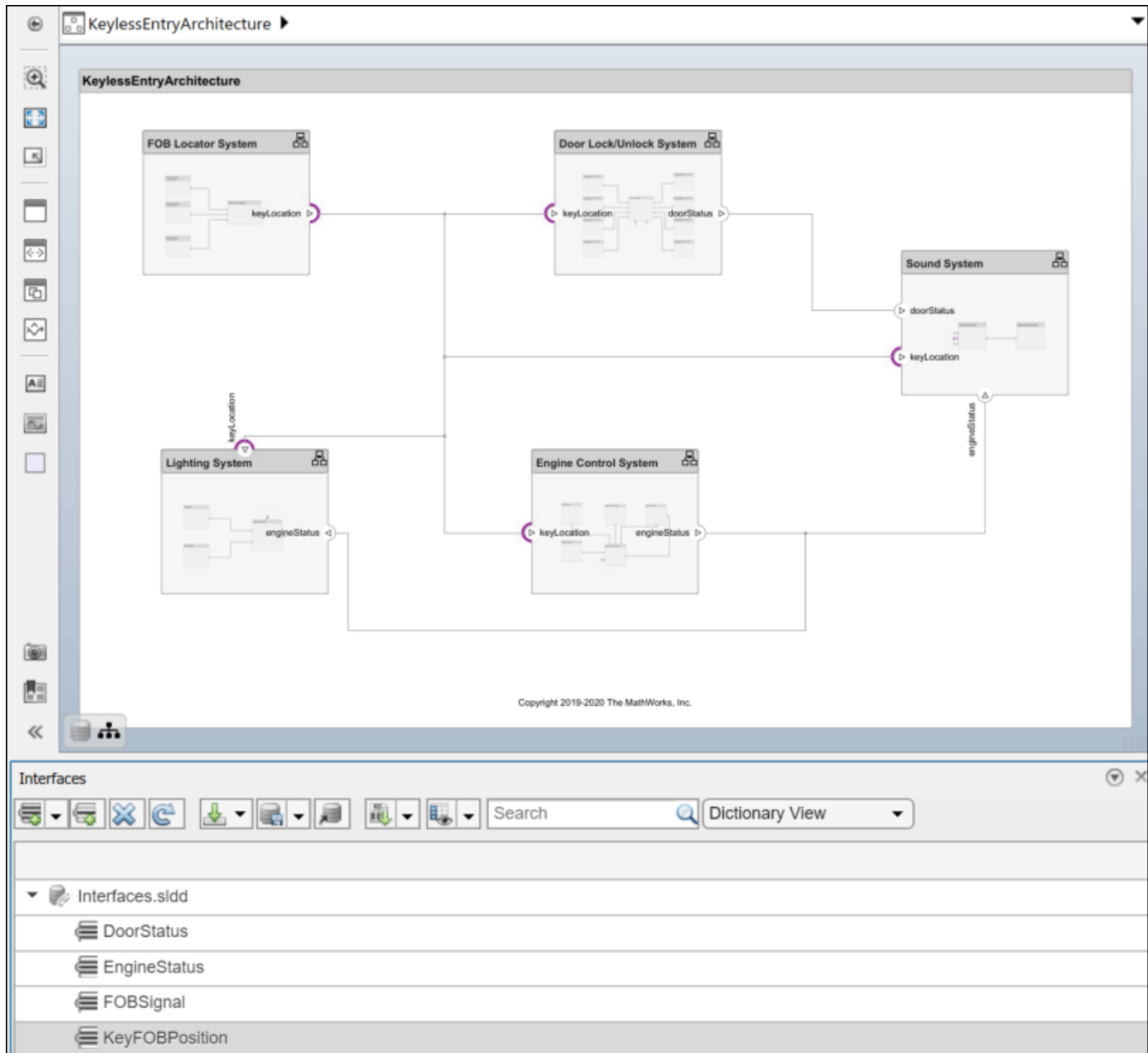
All components in the architecture have the appropriate stereotype applied. Use the Profile Editor tool to define profiles, stereotypes, and properties to apply to components, ports, connectors, and interfaces. Each of the component stereotypes inherits properties from the parent stereotype BaseComponent. For each stereotyped component, you can define the Cost, ReviewStatus, or Latency property values.



To stereotype components and ports, first apply the profile `AutoProfile` to the top model. Each of the component stereotypes includes an icon that represents the usage of the stereotype and indicates the component type. To apply stereotypes to all model elements in a batch process, use the `Apply Stereotypes` dialog. To apply stereotypes one by one, use the `Property Inspector`. For more information, see “Use Stereotypes and Profiles” on page 4-10.

Define Port Interfaces to Describe Data Flow

Composite data interfaces assigned to ports allow you to decompose the data transfer through those ports along the connections between them. For example, the `keyFOBPosition` interface might describe the elements of information passing through the `keyLocation` ports to different components of the architecture.

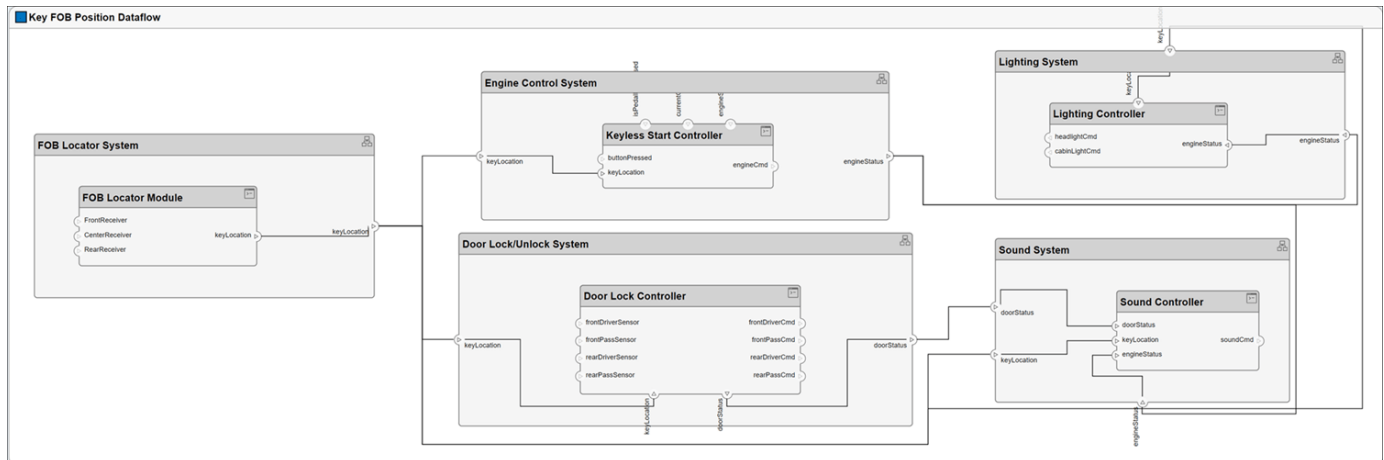


The composite data interfaces that have not been decomposed demonstrate an early version of interfaces design using the Interface Editor. To decompose data interfaces, you can add data elements, or data elements typed by other data interfaces or typed by other value types. For more information, see “Assign Interfaces to Ports” on page 3-10.

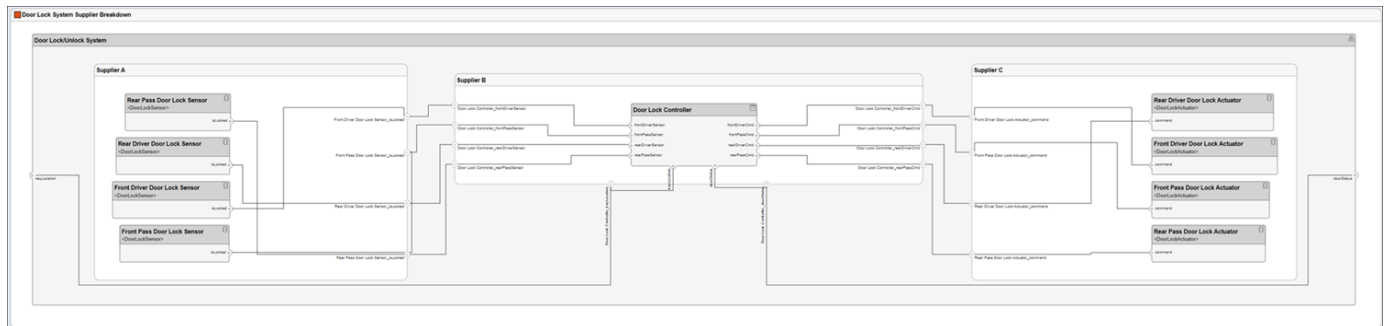
Create Views to Present to Stakeholders

You can create, view, and edit architecture views in the Architecture Views Gallery. To launch the editor, on the **Modeling** tab in the toolstrip, click **Architecture Views**. Filtered views update dynamically with changes to the model. Alternatively, you can use Spotlight Views, which are transient views centered on specific components. For more information, see “Create Spotlight Views” on page 9-2. The KeylessEntryArchitecture model has these views:

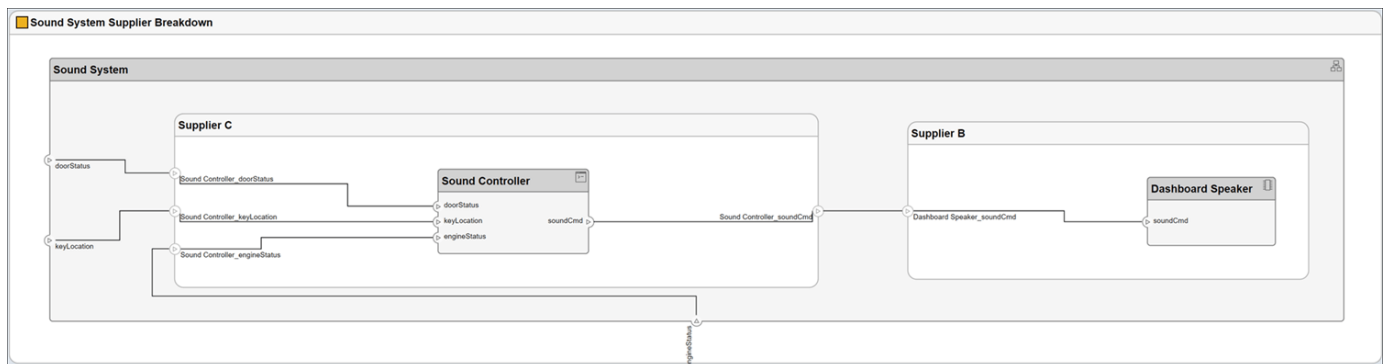
- Key FOB Position Dataflow — An operational view of the components in the model that make use of the KeyFOBPosition interface.



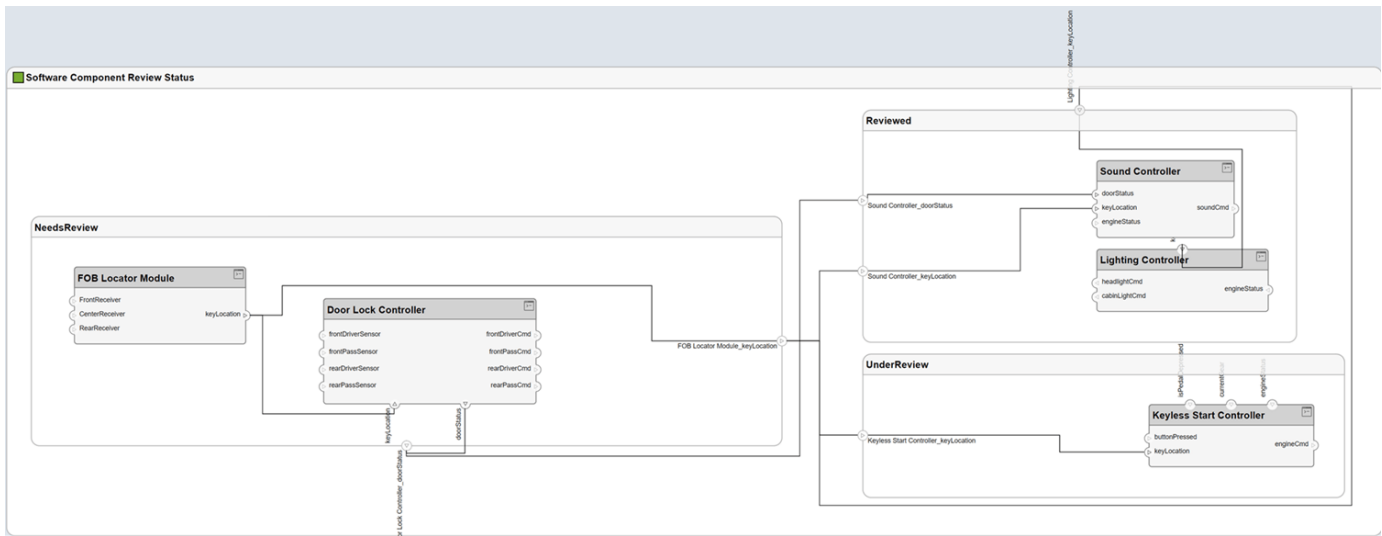
- Door Lock System Supplier Breakdown — A functional view of the components in the door lock system grouped by which supplier provides the given components.



- Sound System Supplier Breakdown — A functional view of the components in the sound system grouped by which supplier provides 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` | `getView` | `openViews` | `deleteView` | `systemcomposer.view.View` | `systemcomposer.view.ElementGroup`

More About

- “Create Architecture Views Interactively” on page 9-5
- “Create Architectural Views Programmatically” on page 9-15
- “Display Component Hierarchy and Architecture Hierarchy Using Views” on page 9-21
- “Organize System Composer Files in Projects” on page 10-2
- “Modeling System Architecture of Small UAV” on page 1-32

Manage Architecture Models

- “Organize System Composer Files in Projects” on page 10-2
- “Compare Model Differences Using System Composer Comparison Tool” on page 10-4

Organize System Composer Files in Projects

Use projects to organize your work, manage files and settings, and interact with source control. Using System Composer generates multiple files, including but not limited to:

- Architecture models (.slx)
- Requirements Toolbox links (.slmx) and requirement sets (.slreqx)
- Allocation sets (.mldatx)
- Profiles (.xml)
- Interface data dictionaries (.slidd)
- Simulink Test files (.mldatx)
- MATLAB functions (.m) and live scripts (.mlx)
- Simulink behavior models (.slx)

To help organize these files, use projects.

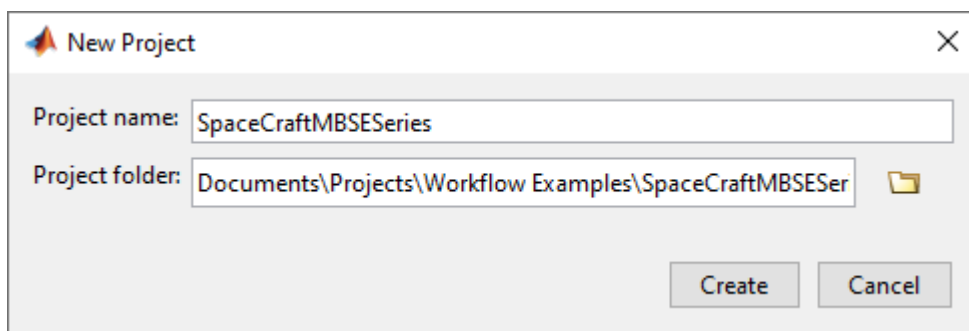
Use Projects to Organize Files and Folders

Create a project from a folder with supporting files and folders.

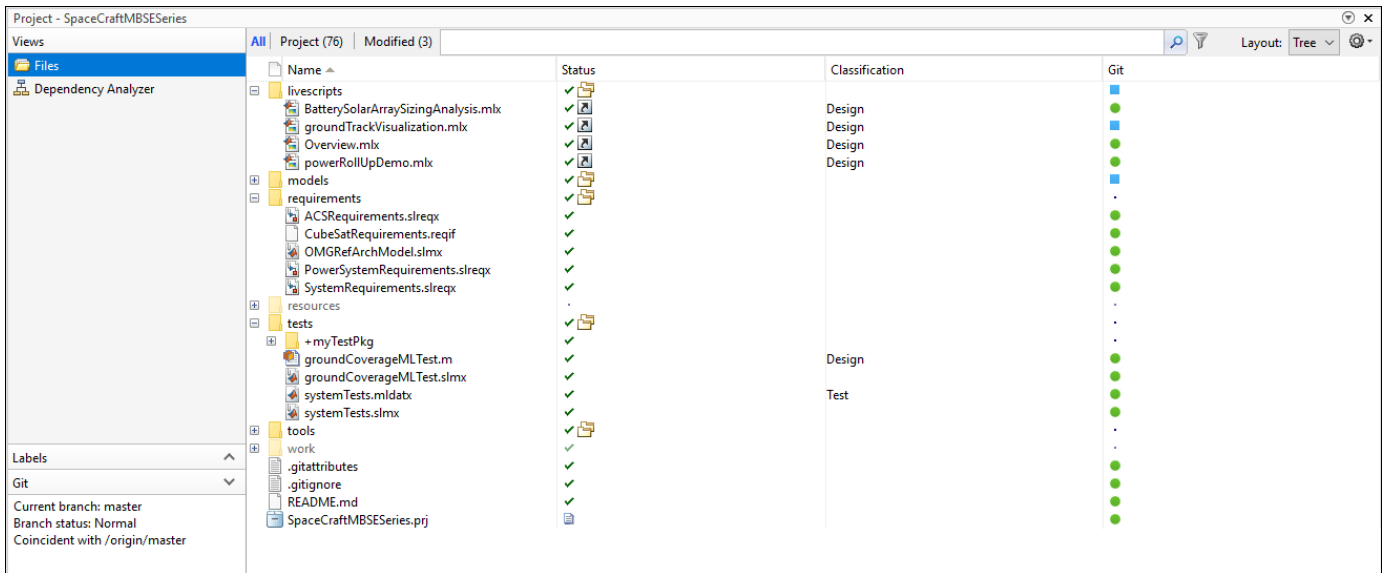
For example, this folder structure represents typical steps in the process of model-based systems engineering: models, profiles, interfaces, requirements, tools, tests, livescripts

The `models` folder can include architecture models, Simulink behavior models, and requirement links. If architecture models and behavior models are constructed separately, you can split the `models` folder into two folders, `architectures` and `simulation`, and decompose the folders further to represent the different stages of architectural model-based design. The `tools` folder can include functions and scripts for trade studies and analyses.

- 1 In MATLAB, navigate to the directory where your model files and artifacts are located.
- 2 Select **New > Project > From Folder**. Enter a name for your project.



- 3 The files in the folder you specify are added to the project, and the **Project** menu appears. To generate your own project shortcuts, on the **Project Shortcuts** tab, click **New Shortcut** or **Organize Groups**.
- 4 You can open the project again using the generated `.prj` file in your directory.



Any changes you make will be organized in the project. You can manage changes to files with multiple contributors using source control. For more information on source control with projects, see “About Source Control with Projects”.

To illustrate file dependencies across the project, use the **Dependency Analyzer**. For more information, see “Dependency Analysis for Projects”. To check and upgrade the project, use the **Run Checks** option.

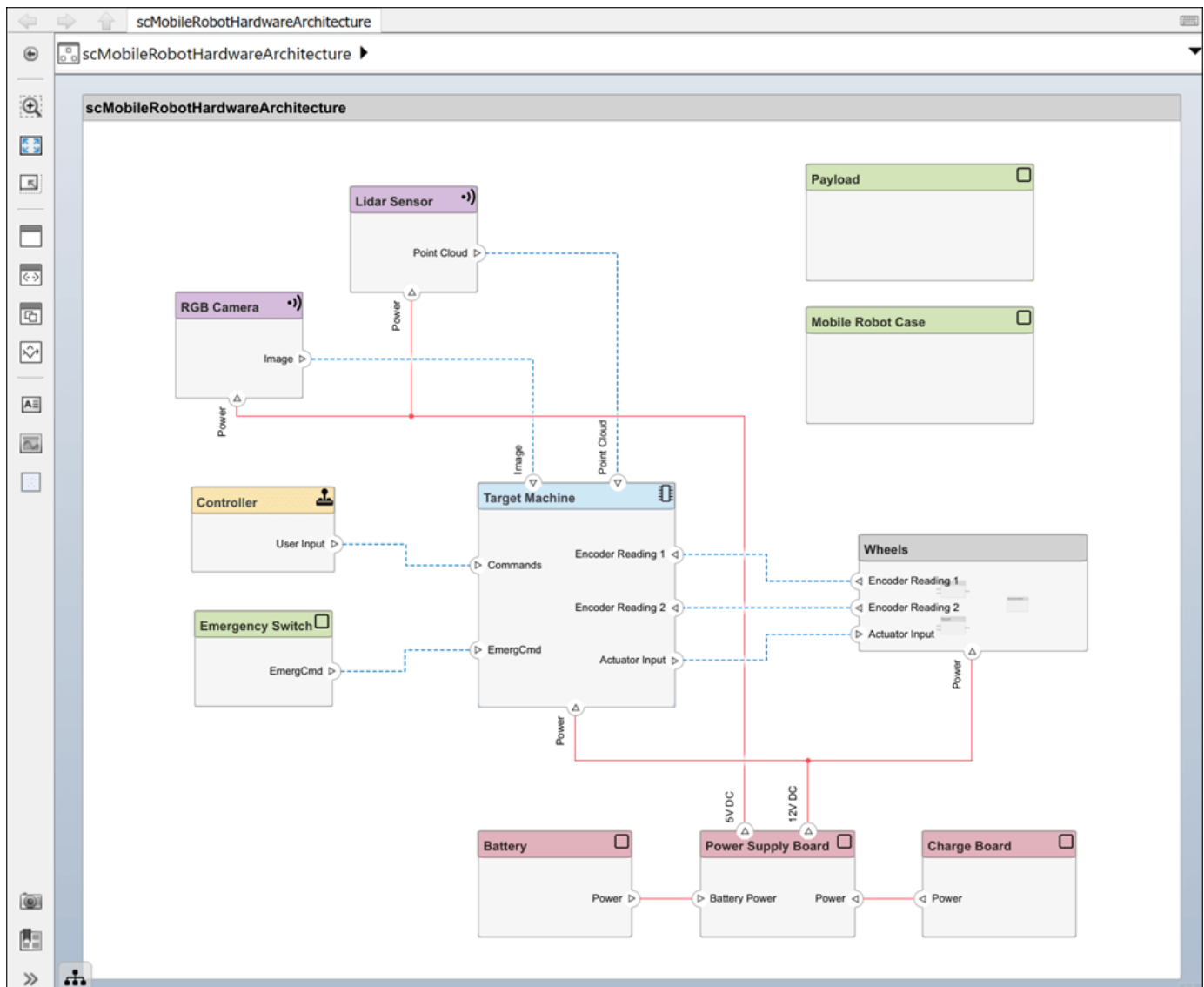
See Also

More About

- “Project Management”
- “Modeling System Architecture of Small UAV” on page 1-32
- “Modeling System Architecture of Keyless Entry System” on page 9-25
- “Allocate Architectures in Tire Pressure Monitoring System” on page 6-5
- “Calculate Endurance Using Quadcopter Architectural Design” on page 7-16
- “Design Insulin Infusion Pump Using Model-Based Systems Engineering” on page 7-23

Compare Model Differences Using System Composer Comparison Tool

This example shows how to use the Comparison Tool to compare two System Composer™ architecture models with differences in architectural data. The models represent a mobile robot hardware architecture and an edited version of the same model.



To open the Comparison Tool, enter this command.

```
visdiff("scMobileRobotHardwareArchitecture.slx", "scMobileRobotHardwareArchitectureEdited.slx")
```

Compare Structural Differences

The first section of the System Composer comparison report is called **Architecture**. The differences in this section include changes to components, ports, and connectors. This section also

includes changes to component types, port types, and any owned interfaces added to ports. The comparison report displays:

- A new physical port named Temperature Control added to the Power Supply Board component and connected to the Battery Pack component.
- A new port named Wheel Stud connected from the Wheels component to the Mobile Robot Case component.
- A new owned interface with elements Signal and Message owned by the Command output port.
- A port on the Target Machine component renamed from Commands to Command.
- The Controller component that is converted to a Stateflow® chart component.

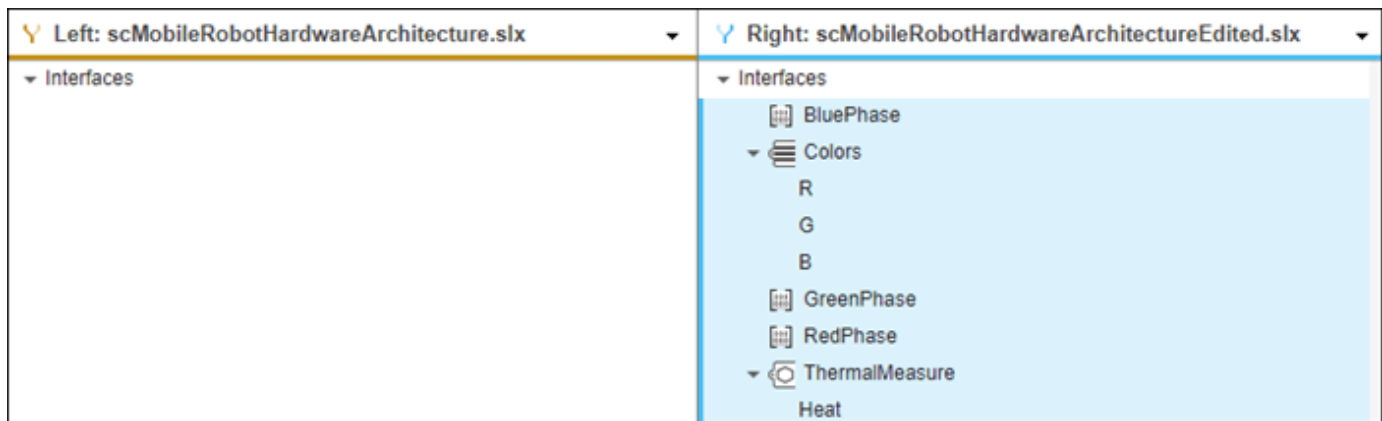
Left: scMobileRobotHardwareArchitecture.slx	Right: scMobileRobotHardwareArchitectureEdited.slx
Architecture	Architecture
<ul style="list-style-type: none"> scMobileRobotHardwareArchitecture <ul style="list-style-type: none"> Power Supply Board Controller <ul style="list-style-type: none"> User Input Wheels <ul style="list-style-type: none"> Actuator Input Power Mobile Robot Case Battery <ul style="list-style-type: none"> Power Target Machine <ul style="list-style-type: none"> Image Commands RGB Camera <ul style="list-style-type: none"> Image 	<ul style="list-style-type: none"> scMobileRobotHardwareArchitectureEdited <ul style="list-style-type: none"> Power Supply Board <ul style="list-style-type: none"> Temperature Control Controller <ul style="list-style-type: none"> User_Input Wheels <ul style="list-style-type: none"> Actuator Input Power Wheel Stud Mobile Robot Case <ul style="list-style-type: none"> Wheel Stud Battery Pack <ul style="list-style-type: none"> Battery Power Temperature Control Target Machine <ul style="list-style-type: none"> Image Command <ul style="list-style-type: none"> Signal Message RGB Camera <ul style="list-style-type: none"> Image
<ul style="list-style-type: none"> Controller : User Input -> Target Machine : Commands 	<ul style="list-style-type: none"> Mobile Robot Case : Wheel Stud -> Wheels : Wheel Stud Conn <ul style="list-style-type: none"> Battery Pack : Temperature Control Power Supply Board : Temperature Control

To learn more about specific changes, you can select the row in the comparison report and view additional information in the bottom pane. As you click on each row of the comparison report, the corresponding open models on the right side are highlighted.

Compare Interface Dictionary Differences

The second section of the System Composer comparison report is called **Interfaces**. The differences in this section include changes to interfaces on the model data dictionary. The comparison report displays:

- Three new value type interfaces called RedPhase, GreenPhase, and BluePhase.
- A new Colors data interface with three data elements: R, G, and B.
- A new physical interface named ThermalMeasure with the physical element Heat.

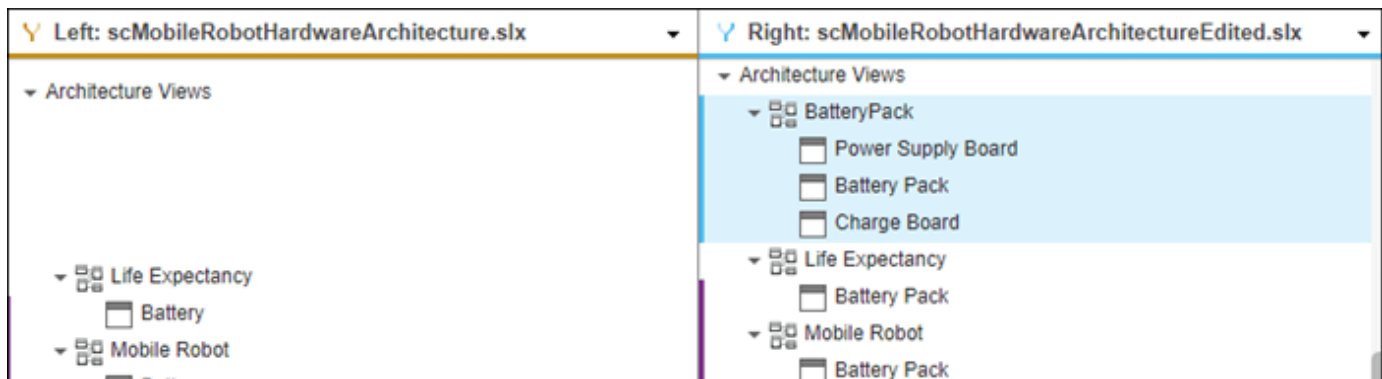


When you click on the data element: R, G, or B, you can see that the element is typed by its value type. You can also inspect the physical domain that types the physical element Heat.

Compare Views Differences

The third section of the System Composer comparison report is called **Architecture Views**. The differences in this section added or deleted architecture views and shows whether the view mode has changed between Component Diagram, Component Hierarchy, or Architecture Hierarchy. The comparison report displays:

- The new BatteryPack view and the components that are displayed within it: Power Supply Board, Battery Pack, and Charge Board.
- Changes to the existing views Life Expectancy and Mobile Robot due to renaming the Battery component to Battery Pack.



When you click on each of the view names, the bottom pane will indicate if there are further modifications to the views.

See Also

visdiff

More About

- “Compose Architecture Visually” on page 1-2
- “Define Port Interfaces Between Components” on page 3-2
- “Define Profiles and Stereotypes” on page 4-2
- “Create Architecture Views Interactively” on page 9-5
- “Describe Component Behavior Using Simulink” on page 5-2
- “Organize System Composer Files in Projects” on page 10-2

Import and Export Architecture Models

- “Import and Export Architectures” on page 11-2
- “Import and Export Architecture Models” on page 11-5
- “Import System Composer Architecture Using ModelBuilder” on page 11-13

Import and Export Architectures

In System Composer™, an architecture is fully defined by three sets of information:

- Component information
- Port information
- Connection information

You can import an architecture into System Composer when this information is defined in or converted into MATLAB® tables.

In this example, the architecture information of a simple unmanned aerial vehicle (UAV) system is defined in a Microsoft® Excel® spreadsheet and is used to create a System Composer architecture model. It also links elements to the specified system level requirement. You can modify the files in this example to import architectures defined in external tools, when the data includes the required information. The example also shows how to export this architecture information from System Composer architecture model to an Excel spreadsheet.

Architecture Definition Data

You can characterize the architecture as a network of components and import by defining components, ports, connections, interfaces and requirement links in MATLAB tables. The `components` table must include name, unique ID, and parent component ID for each component. The spreadsheet can also include other relevant information required to construct the architecture hierarchy for referenced model, and stereotype qualifier names. The `ports` table must include port name, direction, component, and port ID information. Port interface information may also be required to assign ports to components. The `connections` table includes information to connect ports. At a minimum, this table must include the connection ID, source port ID, and destination port ID.

The `systemcomposer.importModel(importModelName)` function:

- Reads stereotype names from the `components` table and loads the profiles
- Creates components and attaches ports
- Creates connections using the connection map
- Sets interfaces on ports
- Links elements to specified requirements (requires a Requirements Toolbox™ license)
- Saves referenced models
- Saves the architecture model

Instantiate adapter class to read from Excel.

```
modelName = "simpleUAVArchitecture";
```

`ImportModelFromExcel` function reads the Excel file and creates the MATLAB tables.

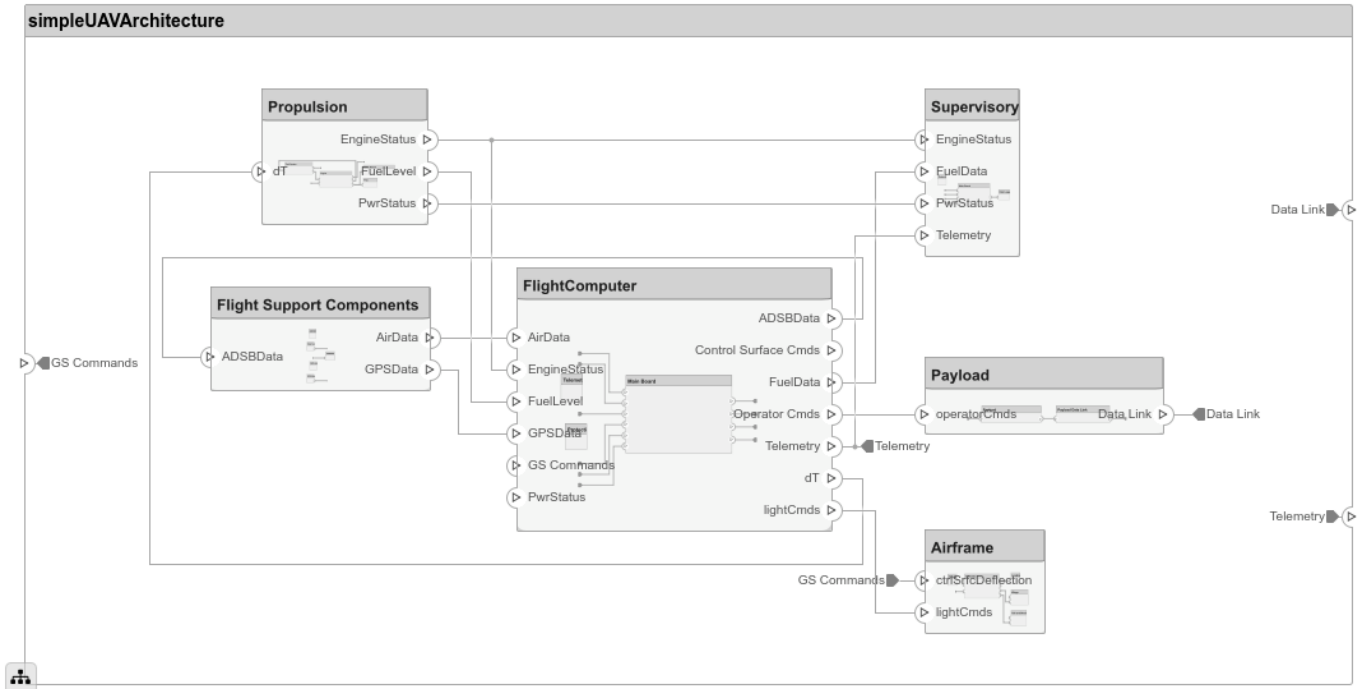
```
importAdapter = ImportModelFromExcel("SmallUAVModel.xls", "Components", ...  
    "Ports", "Connections", "PortInterfaces", "RequirementLinks");  
importAdapter.readTableFromExcel();
```


Import an Architecture

```
model = systemcomposer.importModel(modelName,importAdapter.Components, ...
    importAdapter.Ports,importAdapter.Connections,importAdapter.Interfaces, ...
    importAdapter.RequirementLinks);
```

Auto-arrange blocks in the generated model.

```
Simulink.BlockDiagram.arrangeSystem(modelName)
```



Export an Architecture

You can export an architecture to MATLAB tables and then convert the tables to an external file.

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

The output of the function is a structure that contains the component table, port table, connection table, the interface table, and the requirement links table. Save this structure to an Excel file.

```
SaveToExcel("ExportedUAVModel",exportedSet);
```

See Also

[importModel](#) | [exportModel](#) | [updateLinksToReferenceRequirements](#)

More About

- “Import and Export Architecture Models” on page 11-5
- “Compose Architecture Visually” on page 1-2
- “Decompose and Reuse Components” on page 1-17

- “Manage Requirements” on page 2-8
- “Import System Composer Architecture Using ModelBuilder” on page 11-13
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

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 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 can be one of Input, Output, or Physical.
- 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	Physical	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.
- **Kind** is the kind of connection specified by **Data** by default or **Physical**. The **Kind** column is optional and will default to **Data** if undefined.
- **SourcePortID** is the ID of the source port.
- **DestPortID** is the ID of the destination port.
- **PortIDs** are a comma-separated list of port IDs for physical ports that support nondirectional connections.

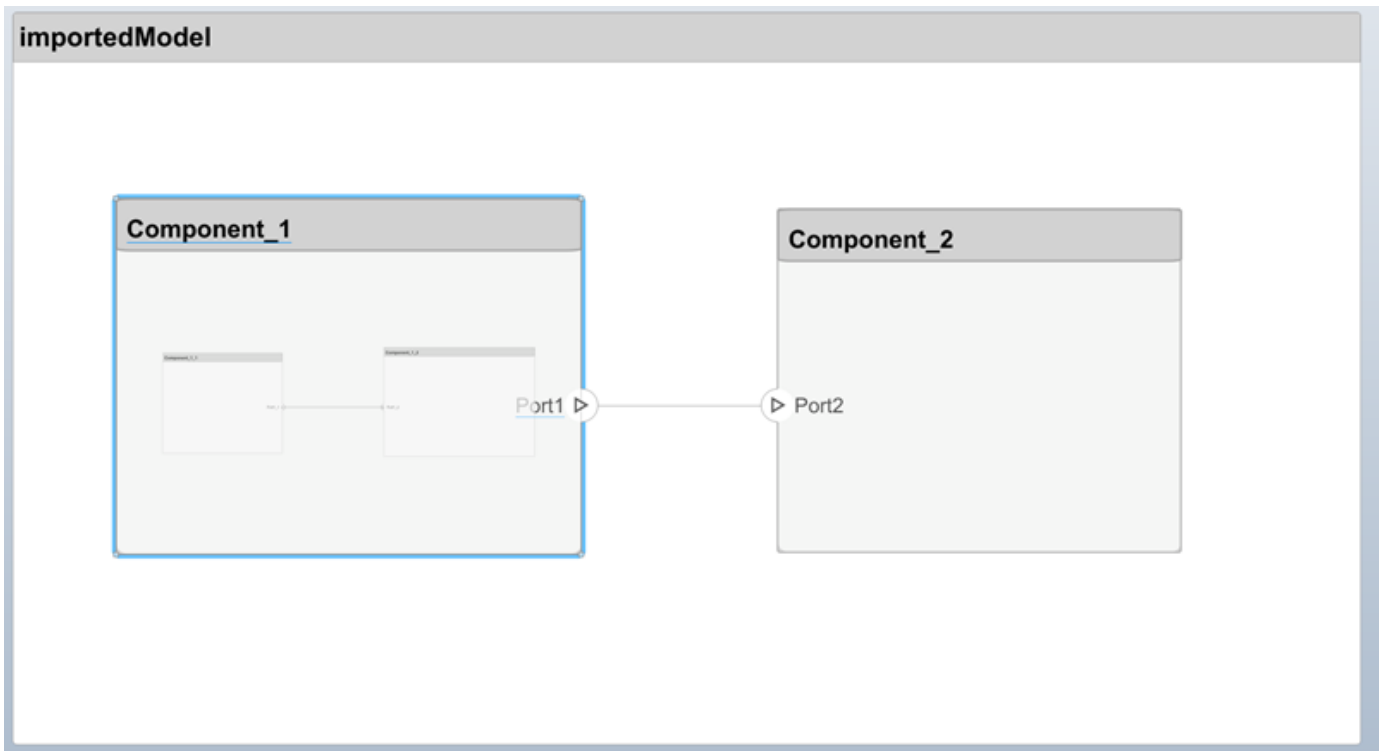
Name	Kind	ID	SourcePortID	DestPortID	PortIDs
Conn1	Data	1	1	2	
Conn2	Physical	2			3,4

Import Basic Architecture

Import the basic architecture from the tables created above into System Composer from the MATLAB Command Window using the `importModel` function.

```
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 Basic Architecture Import

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

Import Data Interfaces and Map Ports to Interfaces

To define the data 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	Data Type	Dimensions	Units	Complexity	Minimum	Maximum
interface 1	1		DataInterface					
elem1	2	1	interface2					
interface 2	3		DataInterface					
elem2	4	1	double	1	""	real	"[]"	"[]"

Name	ID	ParentID	Data Type	Dimensions	Units	Comple xity	Minimu m	Maximu m
elem3	5	1	valueType	3	cm	real	0	100
valueType	6		int32	3	cm	real	0	100
interface3	7		PhysicalInterface					
elec	8	7	Connection: foundation.electrical.electrical					
mech	9	7	Connection: foundation.mechanical.mechanical.rotational					

Data interfaces `interface1` and `interface2` are defined with data elements `elem1` and `elem2` under `interface1`. Data element `elem2` is typed by `interface2`, inheriting its structure. For more information, see “Nest Interfaces to Reuse Data” on page 3-8.

Note Owned interfaces cannot be nested. You cannot define an owned interface as the data type of data elements. For more information, see “Define Owned Interfaces Local to Ports” on page 3-11.

This data interface `interface1` includes a data element `elem3`, which is typed by a value type `valueType` and inherits its properties. For more information, see “Create Value Types as Interfaces” on page 3-7.

This physical interface `interface3` includes physical elements `elec` and `mech`, which are typed under their respective physical domains. For more information, see “Specify Physical Interfaces on Ports” on page 5-76.

To map the added data interface to ports, add the column `InterfaceID` in the `ports` table and specify the data 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

Name	Direction	ID	CompID	InterfaceID
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. For more information, see “Create Variants” on page 1-21.

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	Reference ModelName	ComponentType	ActiveChoice	VariantControl	VariantCondition
root	0						
Component 1	C1	0	simulink_model	Behavior			
VarComp	V2	0		Variant	Choice2		
Choice1	C6	V2				petrol	
Choice2	C7	V2				diesel	
Component 3	C3	0		Stateflow Behavior			
Component 1_1	C4	C1					
Component 1_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. For more information, see “Define Profiles and Stereotypes” on page 4-2.

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 architectural 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.

For more information, see “Manage Requirements” on page 2-8.

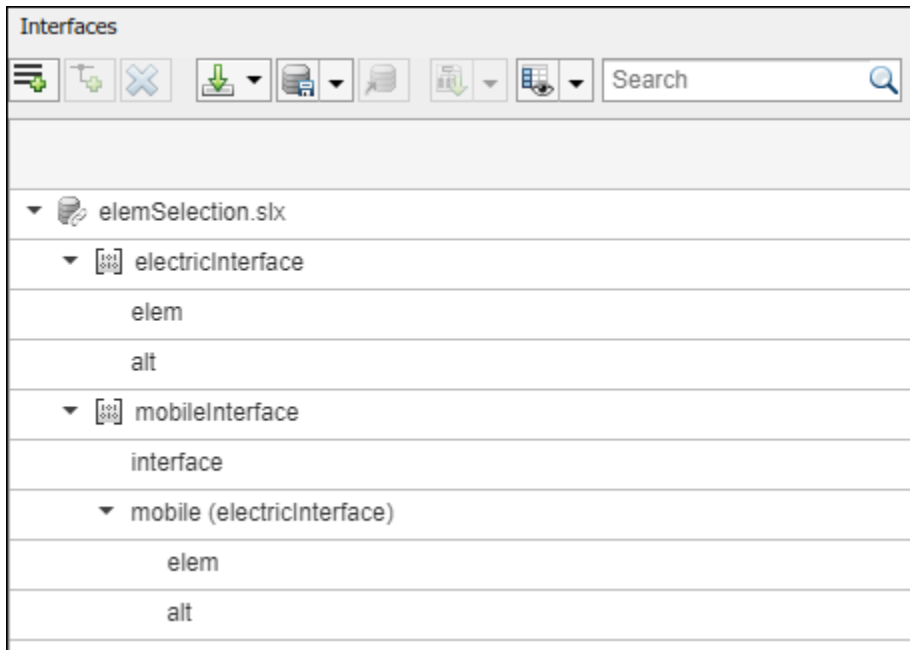
Label	ID	SourceID	DestinationType	DestinationID	Type
rset#1	1	components:1	linktype_rmi_slreq	C:\Temp\rset.slreqx#1	Implement
rset#2	2	components:0	linktype_rmi_slreq	C:\Temp\rset.slreqx#2	Implement
rset#3	3	ports:1	linktype_rmi_slreq	C:\Temp\rset.slreqx#3	Implement
rset#4	4	ports:3	linktype_rmi_slreq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

A Requirements Toolbox 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.



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.

For more information, see “Specify Source Element or Destination Element for Ports” on page 3-14.

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	

Define Architecture Domain for Software Architectures

To specify that the architecture to be imported is a software architecture, specify the domain field of the import structure as "Software". For more information, see “Import and Export Software Architectures” on page 8-5.

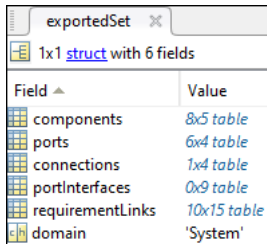
Export Architecture

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

and `requirementLinks`, and the field `domain` that is a character vector that represents the type of architecture being exported. The value of `domain` is 'System' for architecture models or 'Software' for software architecture models.

```
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.



Field ▲	Value
components	8x5 table
ports	6x4 table
connections	1x4 table
portInterfaces	0x9 table
requirementLinks	10x15 table
domain	'System'

If requirements were imported to the model using an external file, in order to export and reimport those requirements, update reference requirement links within the model. You can use the `systemcomposer.updateLinksToReferenceRequirements` function for the requirement links to point to the imported referenced requirements instead of the external documents.

See Also

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

More About

- “Compose Architecture Visually” on page 1-2
- “Decompose and Reuse Components” on page 1-17
- “Describe Component Behavior Using Simulink” on page 5-2
- “Manage Requirements” on page 2-8
- “Import and Export Architectures” on page 11-2
- “Import System Composer Architecture Using ModelBuilder” on page 11-13

Import System Composer Architecture Using ModelBuilder

Import architecture specifications into System Composer™ using the `systemcomposer.io.ModelBuilder` utility class. These architecture specifications can be defined in an external source, such as an Excel® file.

In System Composer, an architecture is fully defined by four sets of information:

- Components and their position in the architecture hierarchy.
- Ports and their mapping to components.
- Connections among components through ports. In this example, we also import interface data definitions from an external source.
- Interfaces in architecture models and their mapping to ports.

This example uses the `systemcomposer.io.ModelBuilder` class to pass all of the above architecture information and import a System Composer model.

In this example, architecture information of a small UAV system is defined in an Excel spreadsheet and is used to create a System Composer architecture model.

External Source Files

- `Architecture.xlsx` — This Excel file contains hierarchical information of the architecture model. This example maps the external source data to System Composer model elements. This information maps in column names to System Composer model elements.

```
# Element      : Name of the element. Either can be component or port name.
# Parent       : Name of the parent element.
# Class        : Can be either component or port(Input/Output direction of the port).
# Domain       : Mapped as component property. Property "Manufacturer" defined in the
                 profile UAVComponent under Stereotype PartDescriptor maps to Domain values in
# Kind         : Mapped as component property. Property "ModelName" defined in the
                 profile UAVComponent under Stereotype PartDescriptor maps to Kind values in
# InterfaceName : If class is of port type. InterfaceName maps to name of the interface link
# ConnectedTo  : In case of port type, it specifies the connection to
                 other port defined in format "ComponentName::PortName".
```

- `DataDefinitions.xlsx` — This Excel file contains interface data definitions of the model. This example assumes this mapping between the data definitions in the Excel source file and interfaces hierarchy in System Composer.

```
# Name         : Name of the interface or element.
# Parent       : Name of the parent interface Name(Applicable only for elements) .
# Datatype     : Datatype of element. Can be another interface in format
                 Bus: InterfaceName
# Dimensions   : Dimensions of the element.
# Units        : Unit property of the element.
# Minimum      : Minimum value of the element.
# Maximum      : Maximum value of the element.
```

Step 1. Instantiate the ModelBuilder Class

You can instantiate the `ModelBuilder` class with a profile name.

```
[stat,fa] = fileattrib(pwd);
if ~fa.UserWrite
```

```
    disp('This script must be run in a writable directory');  
    return;  
end
```

Specify the name of the model to build.

```
modelName = 'scExampleModelBuider';
```

Specify the name of the profile.

```
profile = 'UAVComponent';
```

Specify the name of the source file to read architecture information.

```
architectureFileName = 'Architecture.xlsx';
```

Instantiate the ModelBuilder.

```
builder = systemcomposer.io.ModelBuilder(profile);
```

Step 2. Build Interface Data Definitions

Reading the information in the external source file `DataDefinitions.xlsx` to build the interface data model.

Create MATLAB® tables from the Excel source file.

```
opts = detectImportOptions('DataDefinitions.xlsx');  
opts.DataRange = 'A2';
```

Force `readtable` to start reading from the second row.

```
definitionContents = readtable('DataDefinitions.xlsx',opts);
```

The `systemcomposer.io.IdService` class generates unique ID for a given key.

```
idService = systemcomposer.io.IdService();
```

```
for rowItr =1:numel(definitionContents(:,1))  
    parentInterface = definitionContents.Parent{rowItr};  
    if isempty(parentInterface)
```

In the case of interfaces, add the interface name to the model builder.

```
        interfaceName = definitionContents.Name{rowItr};
```

Get the unique interface ID.

`getID(container, key)` generates or returns (if key is already present) same value for input key within the container.

```
        interfaceID = idService.getID('interfaces',interfaceName);
```

Use `builder.addInterface` to add the interface to the data dictionary.

```
        builder.addInterface(interfaceName,interfaceID);  
    else
```

In the case of an element, read the element properties and add the element to the parent interface.

```

elementName = definitionContents.Name{rowItr};
interfaceID = idService.getID('interfaces',parentInterface);

```

The ElementID is unique within a interface. Append E at the start of an ID for uniformity. The generated ID for an input element is unique within parent interface name as a container.

```

elemID = idService.getID(parentInterface,elementName,'E');

```

Set the data type, dimensions, units, minimum, and maximum properties of the element.

```

datatype = definitionContents.DataType{rowItr};
dimensions = string(definitionContents.Dimensions(rowItr));
units = definitionContents.Units(rowItr);

```

Make sure that input to builder utility function is always a string.

```

if ~ischar(units)
    units = '';
end
minimum = definitionContents.Minimum{rowItr};
maximum = definitionContents.Maximum{rowItr};

```

Use builder.addElementInInterface to add an element with properties in the interface.

```

builder.addElementInInterface(elementName,elemID,interfaceID,datatype,dimensions,units,'E');
end
end

```

Step 3. Build Architecture Specifications

Architecture specifications are created by MATLAB tables from the Excel source file.

```

excelContents = readtable(architectureFileName);

```

Iterate over each row in the table.

```

for rowItr =1:numel(excelContents(:,1))

```

Read each row of the Excel file and columns.

```

class = excelContents.Class(rowItr);
Parent = excelContents.Parent(rowItr);
Name = excelContents.Element{rowItr};

```

Populate the contents of the table.

```

if strcmp(class,'component')
    ID = idService.getID('comp',Name);

```

The Root ID is by default set as zero.

```

if strcmp(Parent,'scExampleSmallUAV')
    parentID = "0";
else
    parentID = idService.getID('comp',Parent);
end

```

Use builder.addComponent to add a component.

```

builder.addComponent(Name,ID,parentID);

```

Read the property values.

```
kind = excelContents.Kind{rowItr};
domain = excelContents.Domain{rowItr};
```

Use `builder.setComponentProperty` to set stereotype and property values.

```
builder.setComponentProperty(ID, 'StereotypeName', 'UAVComponent.PartDescriptor', 'ModelName');
else
```

In this example, concatenation of the port name and parent component name is used as key to generate unique IDs for ports.

```
portID = idService.getID('port',strcat(Name,Parent));
```

For ports on root architecture, the `compID` is assumed as 0.

```
if strcmp(Parent,'scExampleSmallUAV')
    compID = "0";
else
    compID = idService.getID('comp',Parent);
end
```

Use `builder.addPort` to add a port.

```
builder.addPort(Name,class,portID,compID );
```

The `InterfaceName` specifies the name of the interface linked to the port.

```
interfaceName = excelContents.InterfaceName{rowItr};
```

Get the interface ID.

`getID` will return the same IDs already generated while adding interface in Step 2.

```
interfaceID = idService.getID('interfaces',interfaceName);
```

Use `builder.addInterfaceToPort` to map interface to port.

```
builder.addInterfaceToPort(interfaceID,portID);
```

Read the `ConnectedTo` information to build connections between components.

```
connectedTo = excelContents.ConnectedTo{rowItr};
```

`ConnectedTo` is in the format:

```
(DestinationComponentName::DestinationPortName)
```

For this example, consider the current port as source of the connection.

```
if ~isempty(connectedTo)
    connID = idService.getID('connection',connectedTo);
    splits = split(connectedTo, '::');
```

Get the port ID of the connected port.

In this example, port ID is generated by concatenating the port name and the parent component name. If the port ID is already generated, the `getID` function returns the same ID for the input key.

```
connectedPortID = idService.getID('port',strcat(splits(2),splits(1)));
```

Populate the connection table.

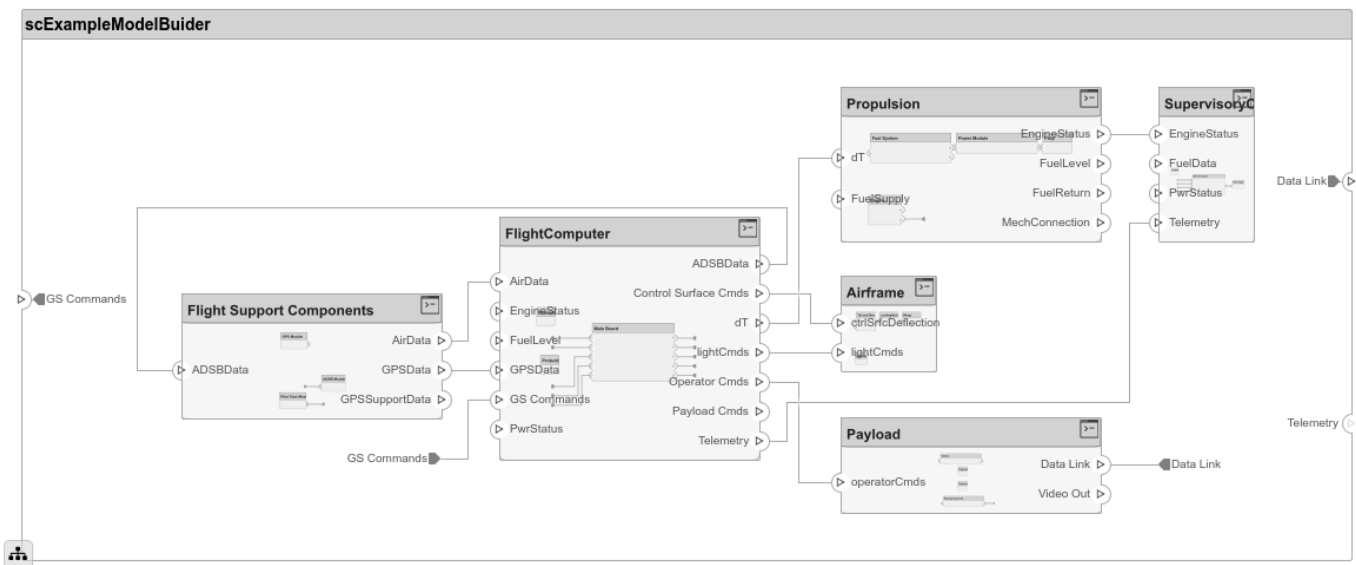
```
sourcePortID = portID;
destPortID = connectedPortID;
```

Use builder.addConnection to add connections.

```
builder.addConnection(connectedTo,connID,sourcePortID,destPortID);
end
end
end
```

Step 3. Import Model from Populated Tables with builder.build Function

```
[model,importReport] = builder.build(modelName);
```



See Also

[systemcomposer.io.ModelBuilder | importModel | exportModel](#)

More About

- “Import and Export Architecture Models” on page 11-5
- “Compose Architecture Visually” on page 1-2
- “Import and Export Architectures” on page 11-2
- “Simulate Mobile Robot with System Composer Workflow” on page 4-22

