System Composer™

User's Guide

MATLAB® & SIMULINK®

R2020b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

System Composer™ User's Guide

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)

Contents

1 Architecture Model Editing

2 Requirements

3 Interface Management

4 Define Architectural Properties

5 Use Simulink Models with System Composer

6 Analyze Architecture Model

Architecture Model Editing

Compose Architecture Visually

In this section...

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

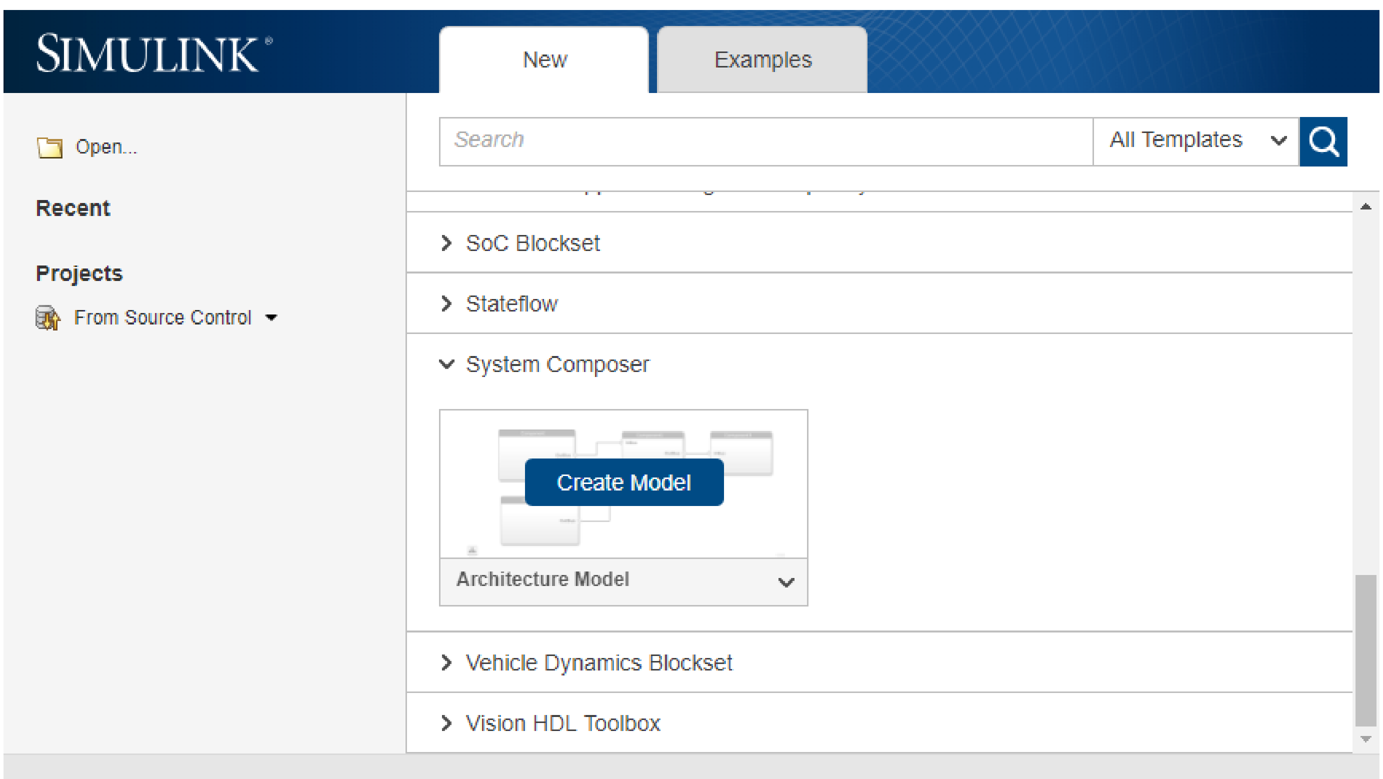
Create an Architecture Model

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

- At the command line, type

  ```
  systemcomposer
  ```

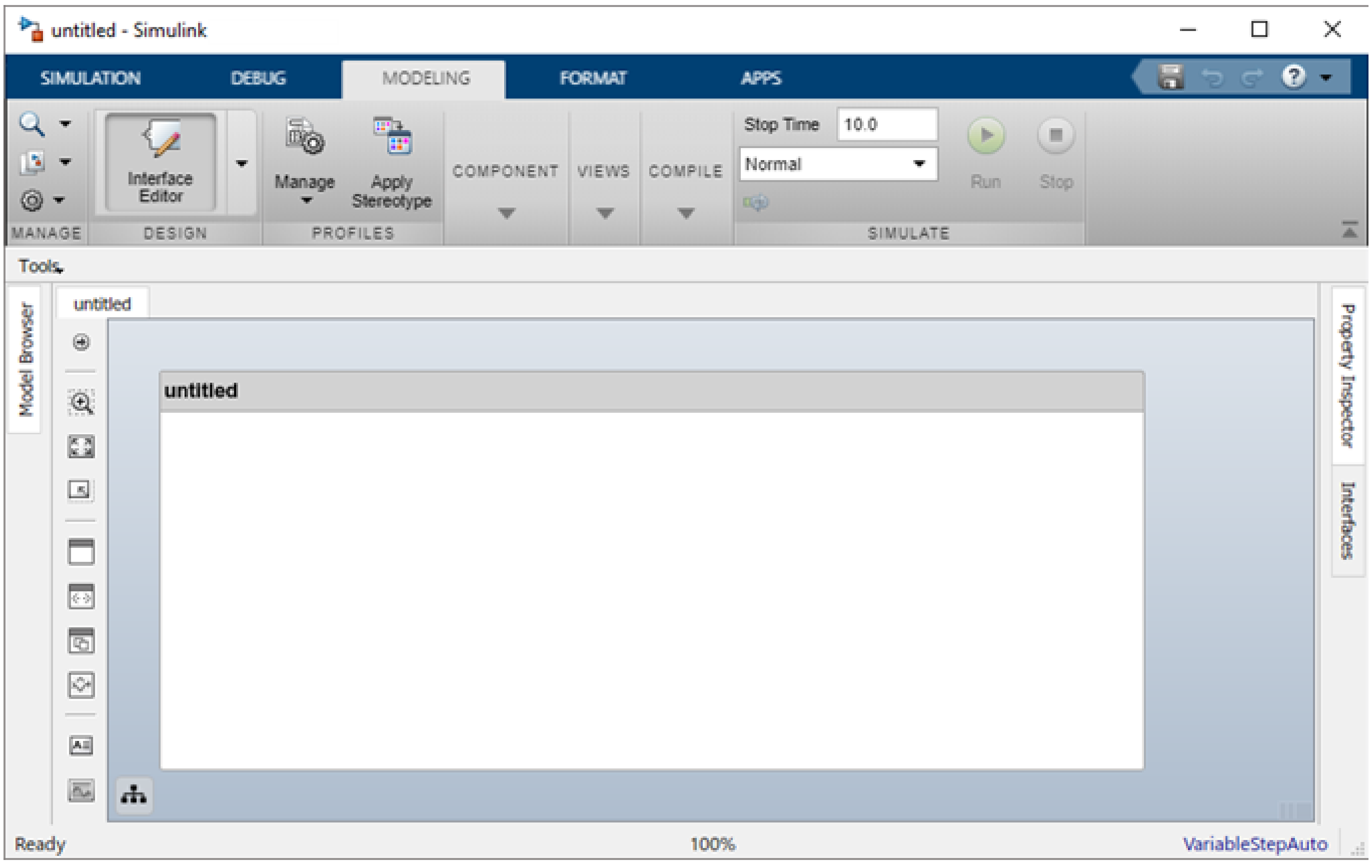
  Select **Architecture Model**.



- From a Simulink model or a System Composer architecture model. On the Simulation tab, select New, and then select Architecture.
- At the MATLAB command line, type:

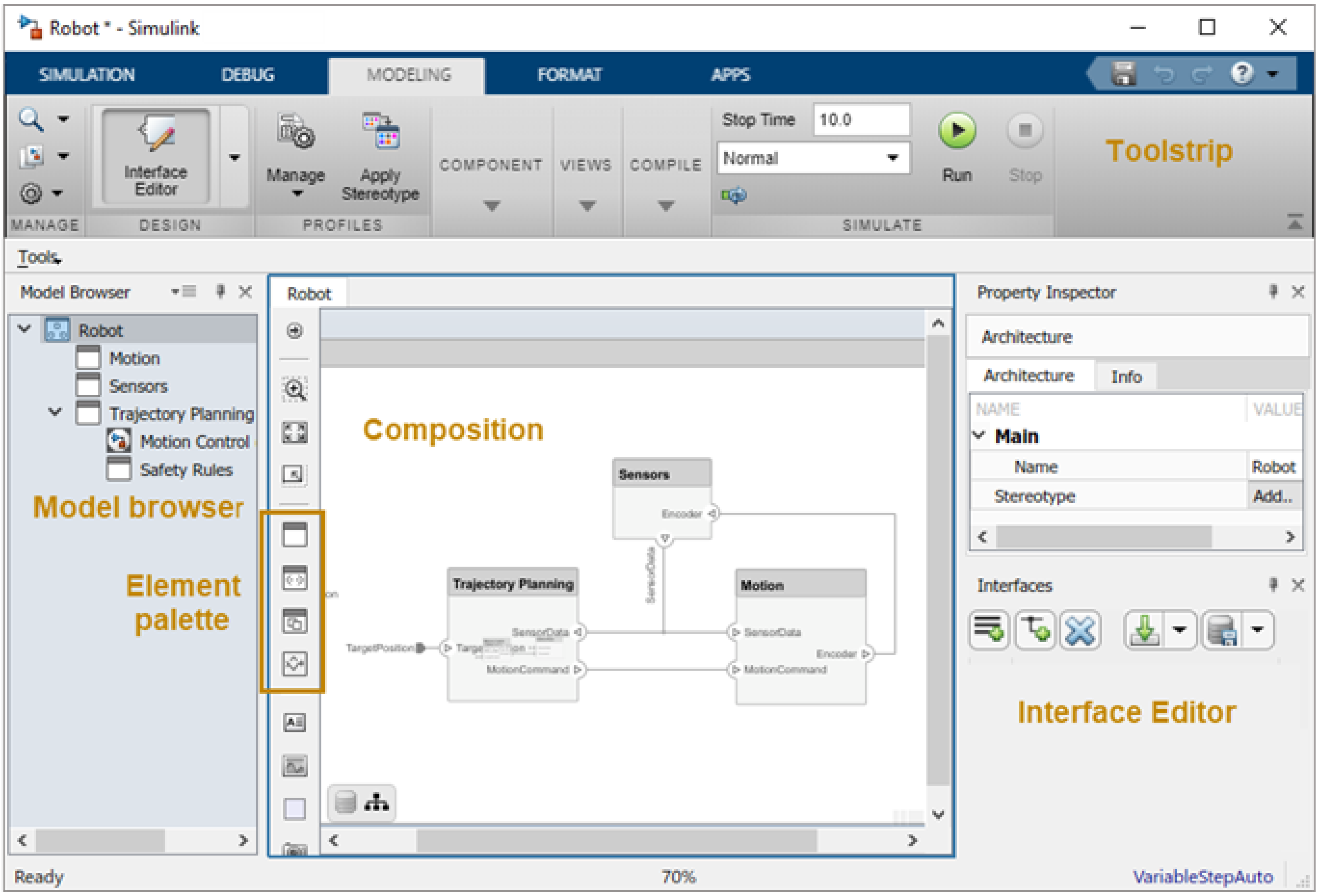
```
archModel = new_system('ModelName','Architecture');
open_system(archModel)
```

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



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

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



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

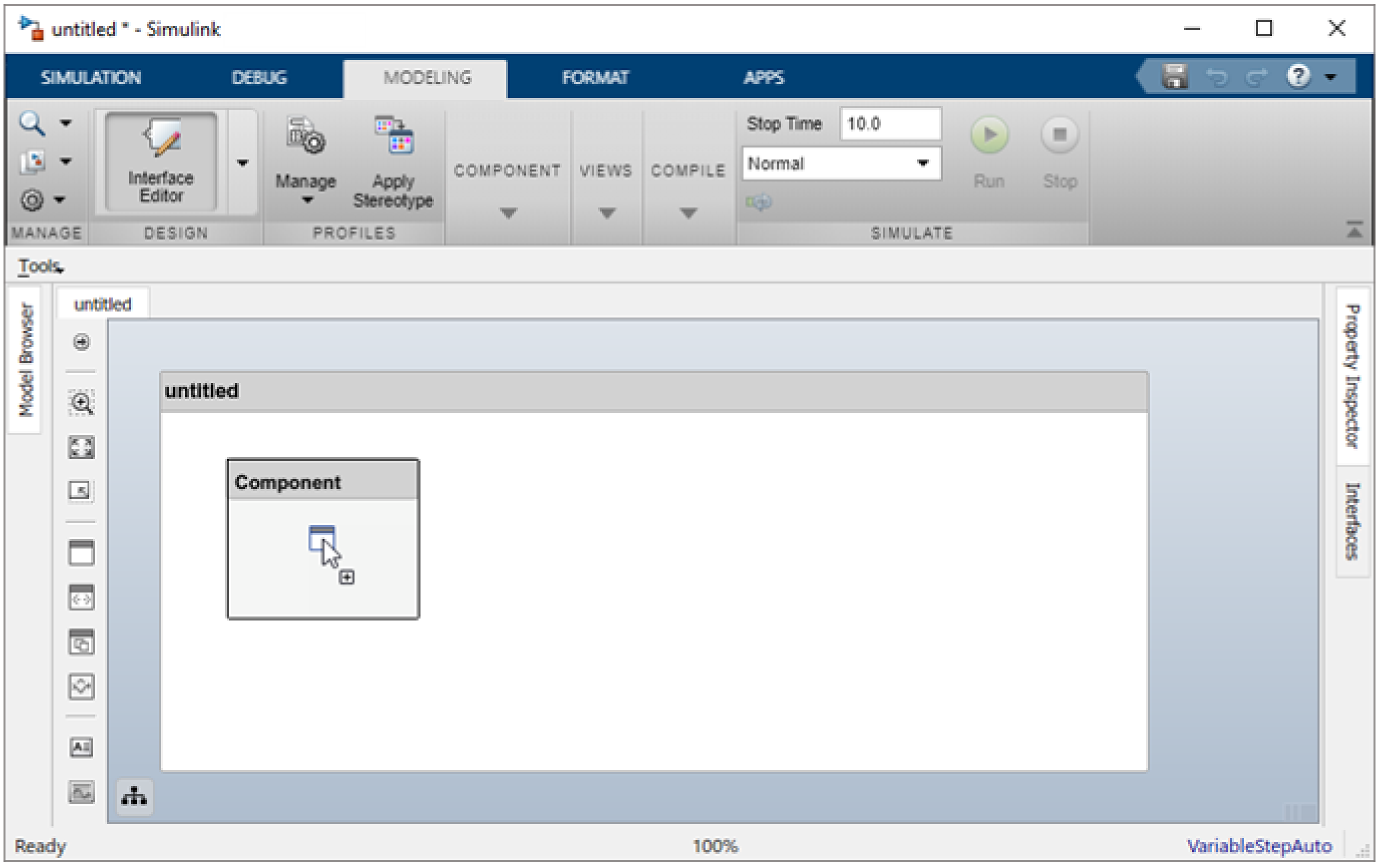
Components

A component is a nontrivial, nearly-independent, and replaceable part of a system that fulfills a clear function in the context of an architecture. 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.

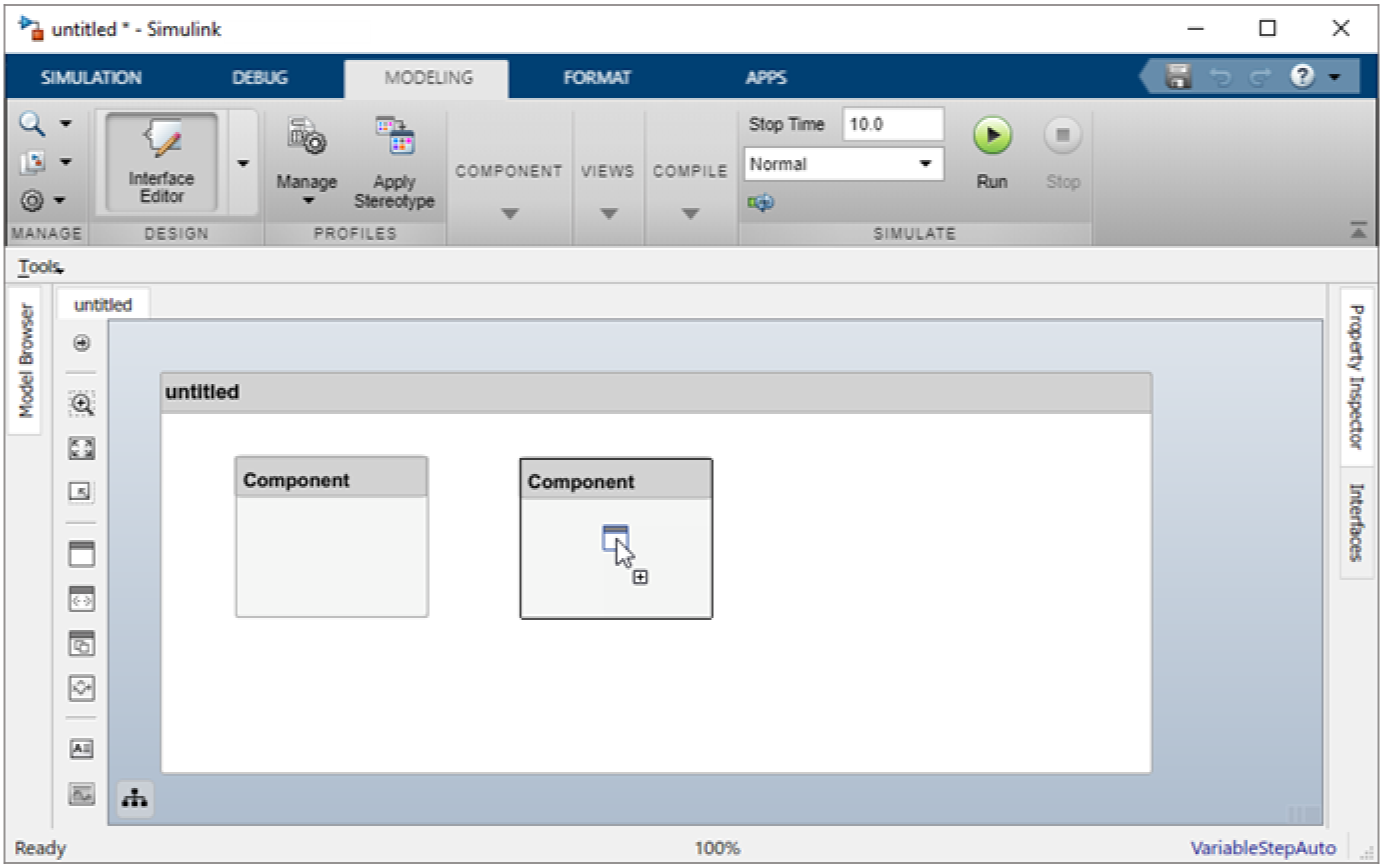
Add Components

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. Click the light blue outline to commit.



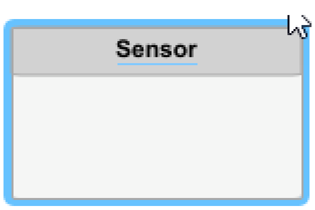
- Create a single component from the palette —

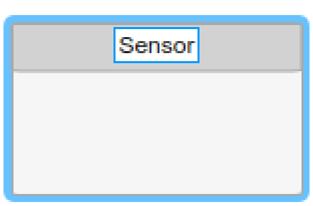


- Create multiple components from the palette —

Name a Component

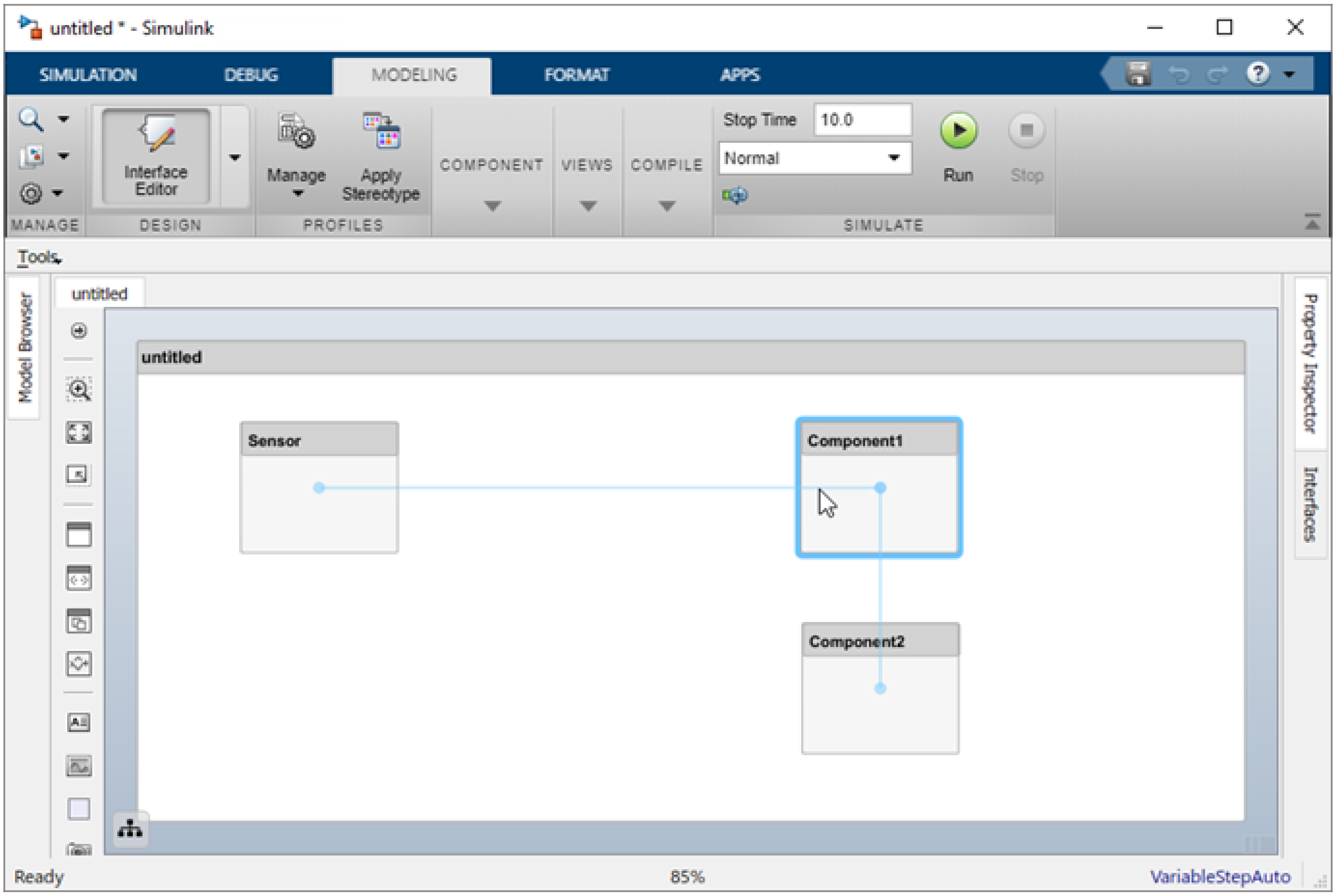
Each component must have a name that is unique within the same architecture level. The name of the component is highlighted upon creation so you can directly type the name. To change the name of a component, click the component and then click its name.





Move a Component

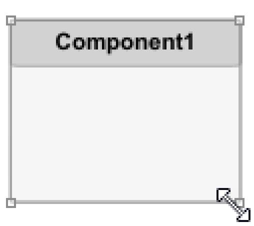
Move a component simply by clicking and dragging it. Blue guidelines may appear to help align the component with other components.



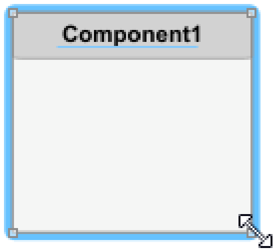
Resize a Component

Resize a component by dragging corners.

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



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



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

Delete a Component

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

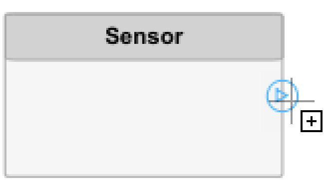
Ports

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

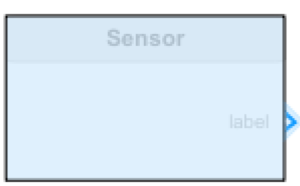
Add a Component Port

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

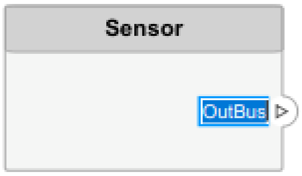
1 Pause over the side of a component. A + sign and a port outline appear.



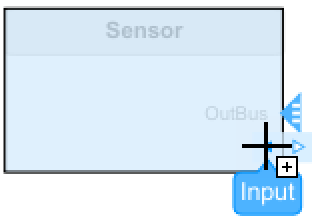
2 Click the port outline. The component is shaded blue and a port arrow appears.



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



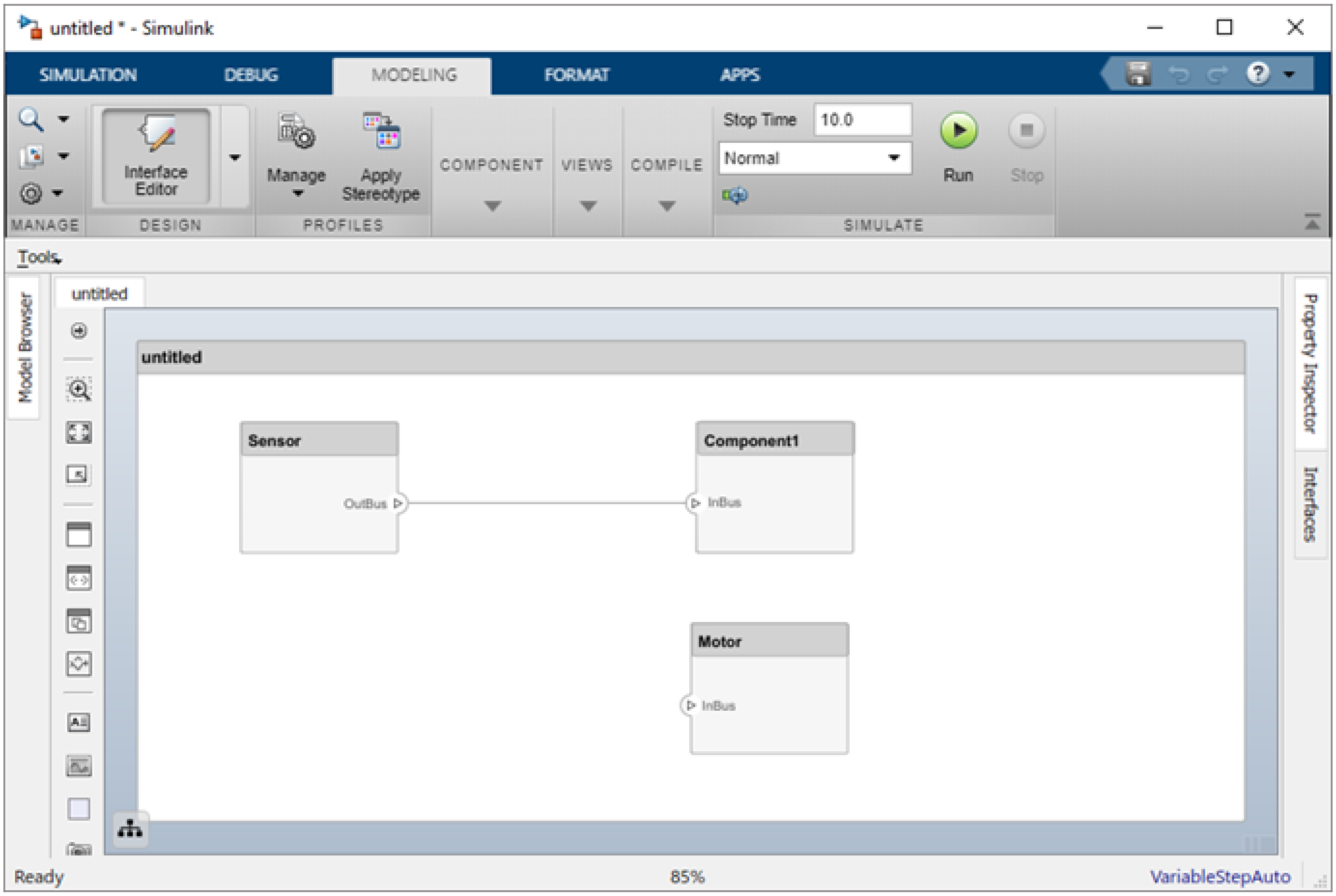
An output port is shown with the  icon and an input port is shown with the  icon. By default, a port created on the top or left edge of a component is an input port, and a port created on the bottom or right edge is an output port. To designate port direction at creation, after you click the edge, pause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



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

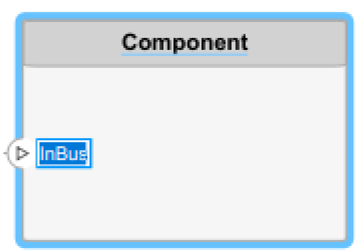
Add an Architecture Port

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



Name a Port

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



Ports of a component must have unique names.

Move a Port

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

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

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

Delete a Port

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

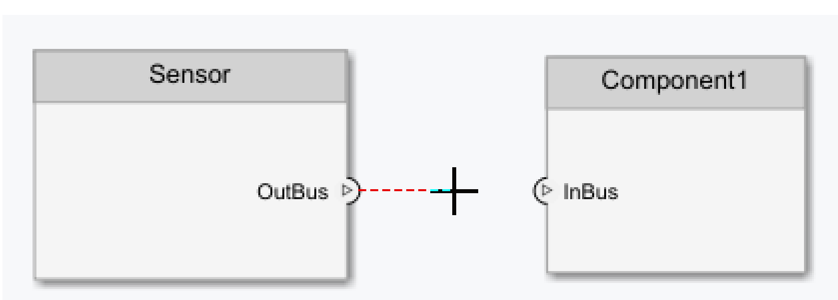
Connections

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

Connect Existing Ports

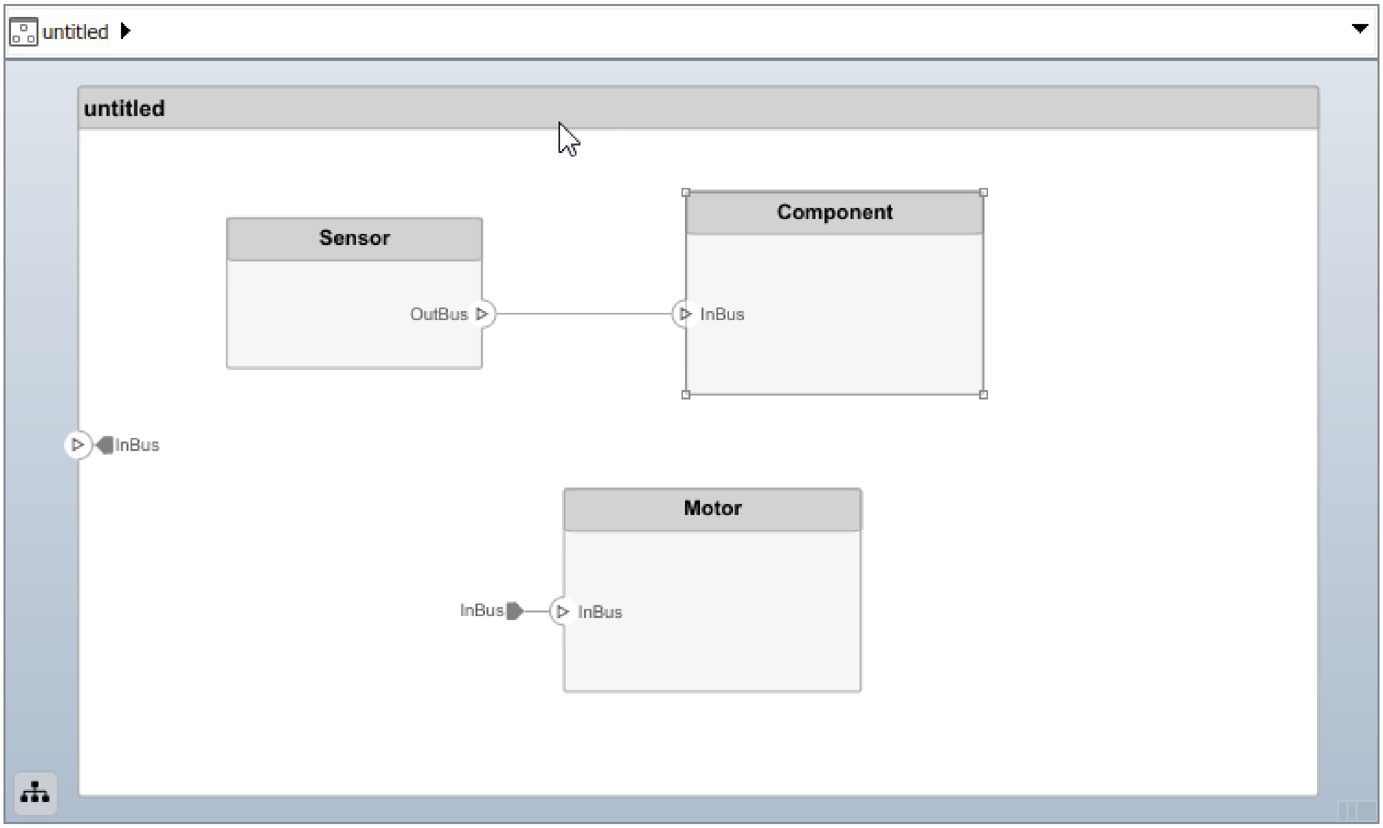
Connect two ports by dragging a line:

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



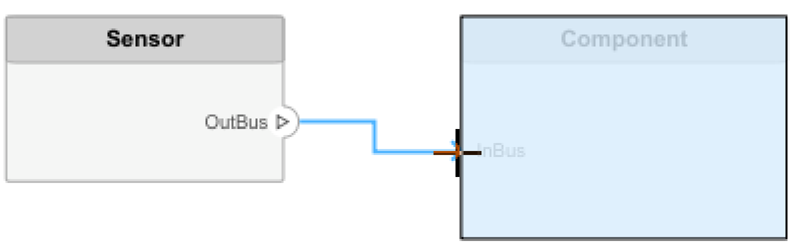
You can take these steps in both directions — input port to output port, or output port to input port. You cannot connect ports that have the same direction.

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



Connect Components Without Ports

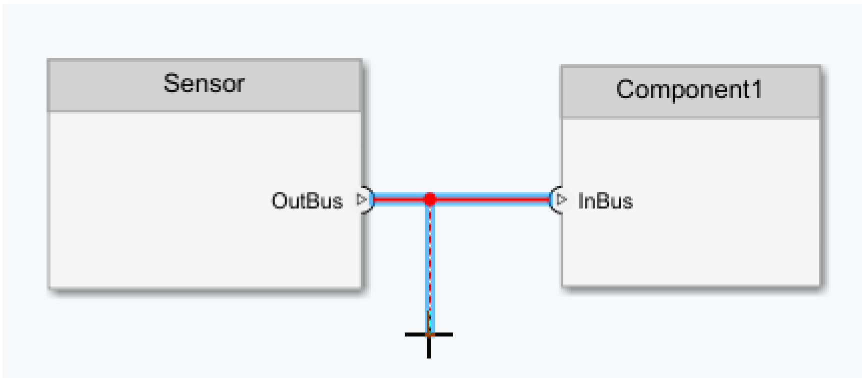
To quickly create ports and connections at the same time, drag a line from one component edge to another. The direction of this connection depends on which edges of the components are used - left and top edges are considered inputs, right and bottom edges are considered outputs. You can also perform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



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

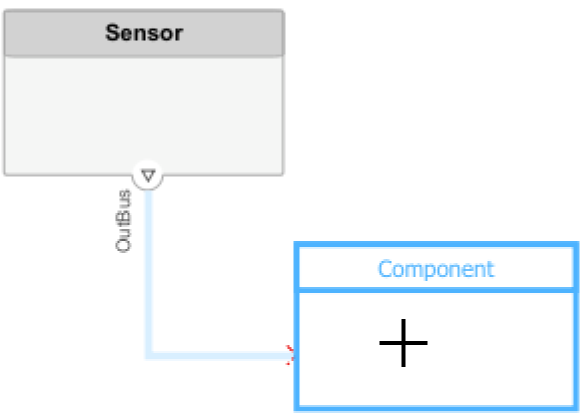
Branch Connections

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



Create New Components Through Connections

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



Importing Architectures

By combining the programmatic APIs of System Composer with MATLAB® support for loading and parsing many different file and databased formats, you can import external Architecture descriptions into System Composer. You can setup a profile with Stereotypes ahead of time to capture the Architecture properties represented in such descriptions. Subsequently, you can use MATLAB programming to create and customize the various Architectural elements through the set of programmatic APIs.

See Also

More About

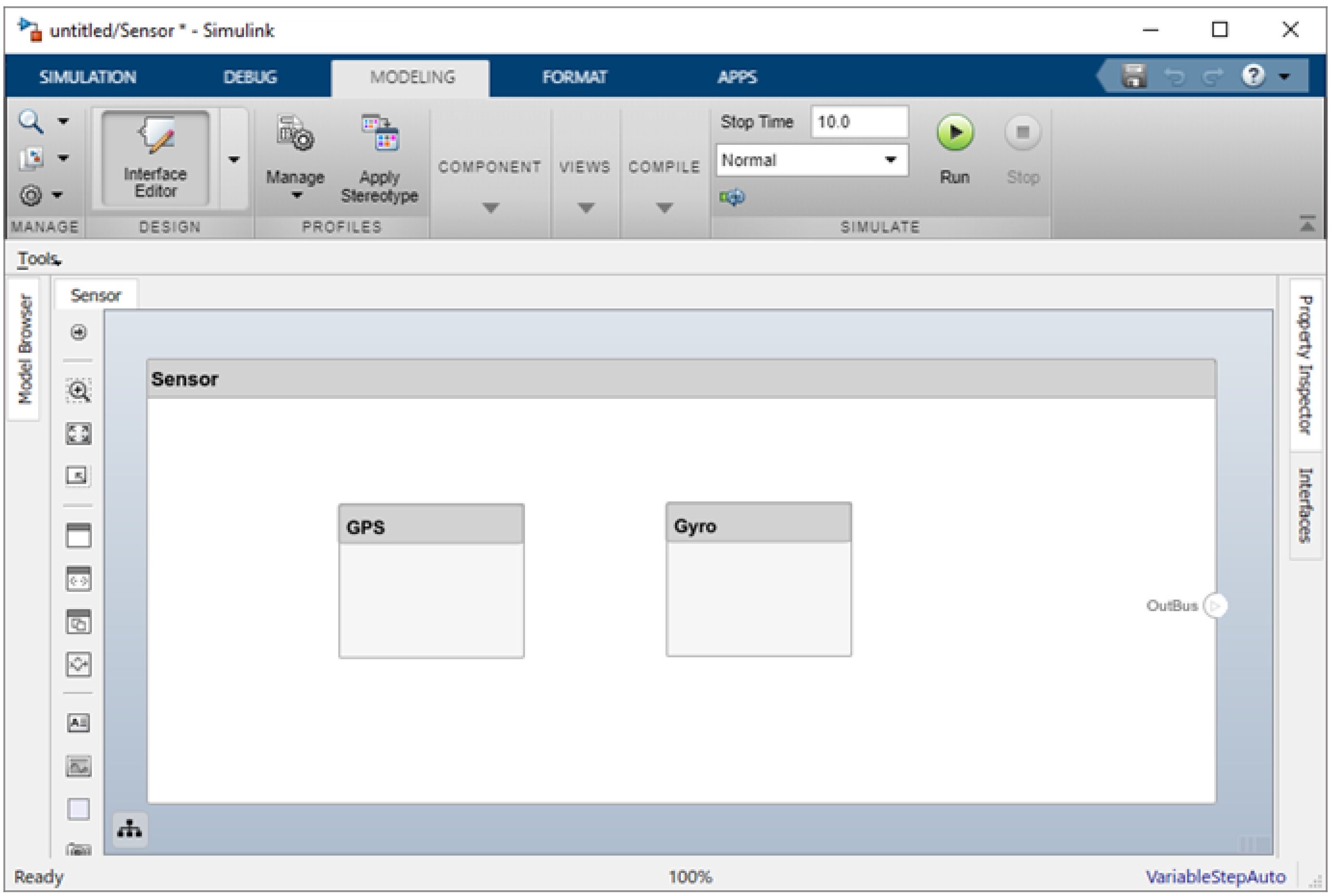
- "Decompose and Reuse Components" on page 1-15
- "Define Interfaces" on page 3-2
- "Assign Interfaces to Ports" on page 3-5

Decompose and Reuse Components

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

Decompose a Component

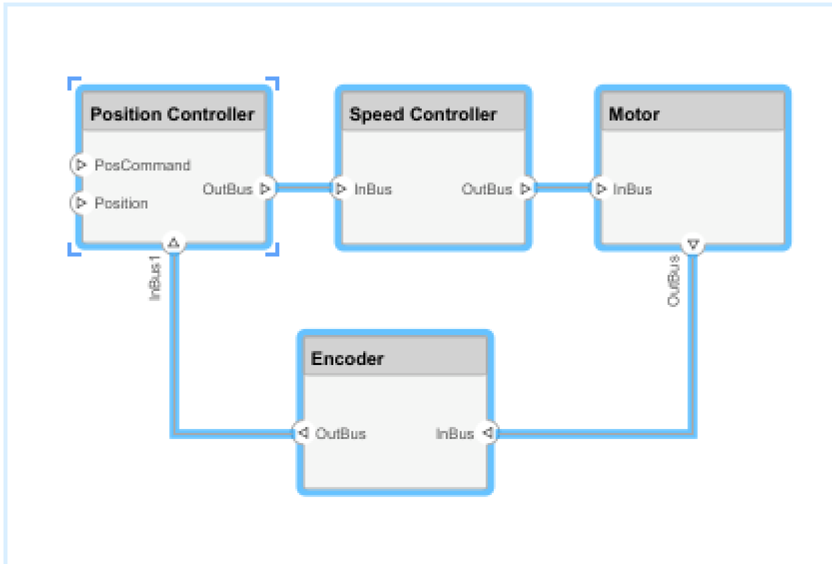
A component can have its own architecture. Double-click a component to view or edit its architecture. When you view the component at this level, its ports appear as architecture ports. You can use the navigation arrows  on the toolbar to move through the hierarchy. Use the Model Browser to view component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



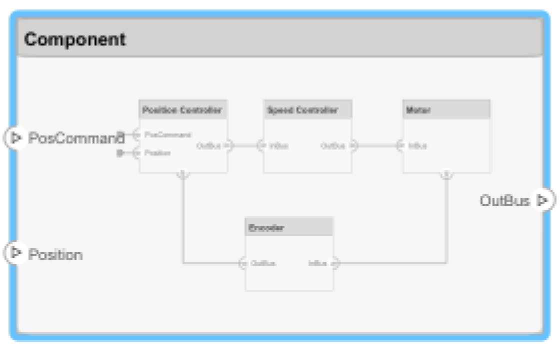
You can add components, ports, and connections at this level to define the architecture.

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

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



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



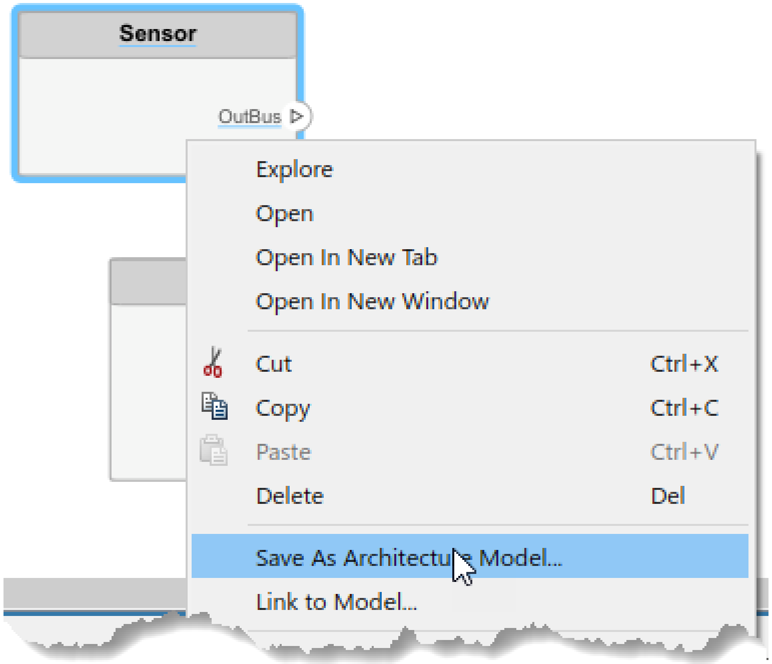
As a result, the new component has the selected components, their ports, and connections as part of its architecture. Any unconnected ports and connections to components outside of the selection become ports on the new component.

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

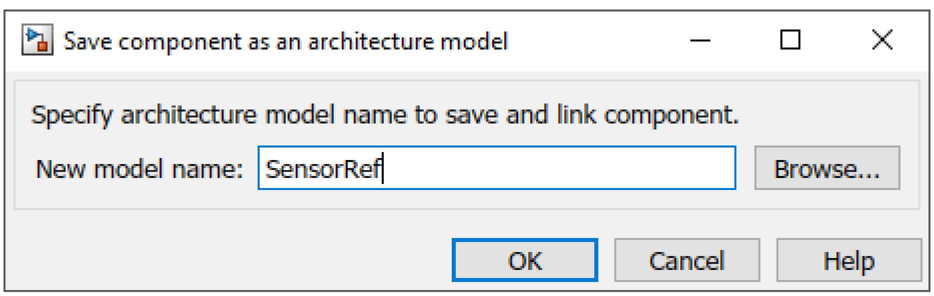
Create a Reference Architecture

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

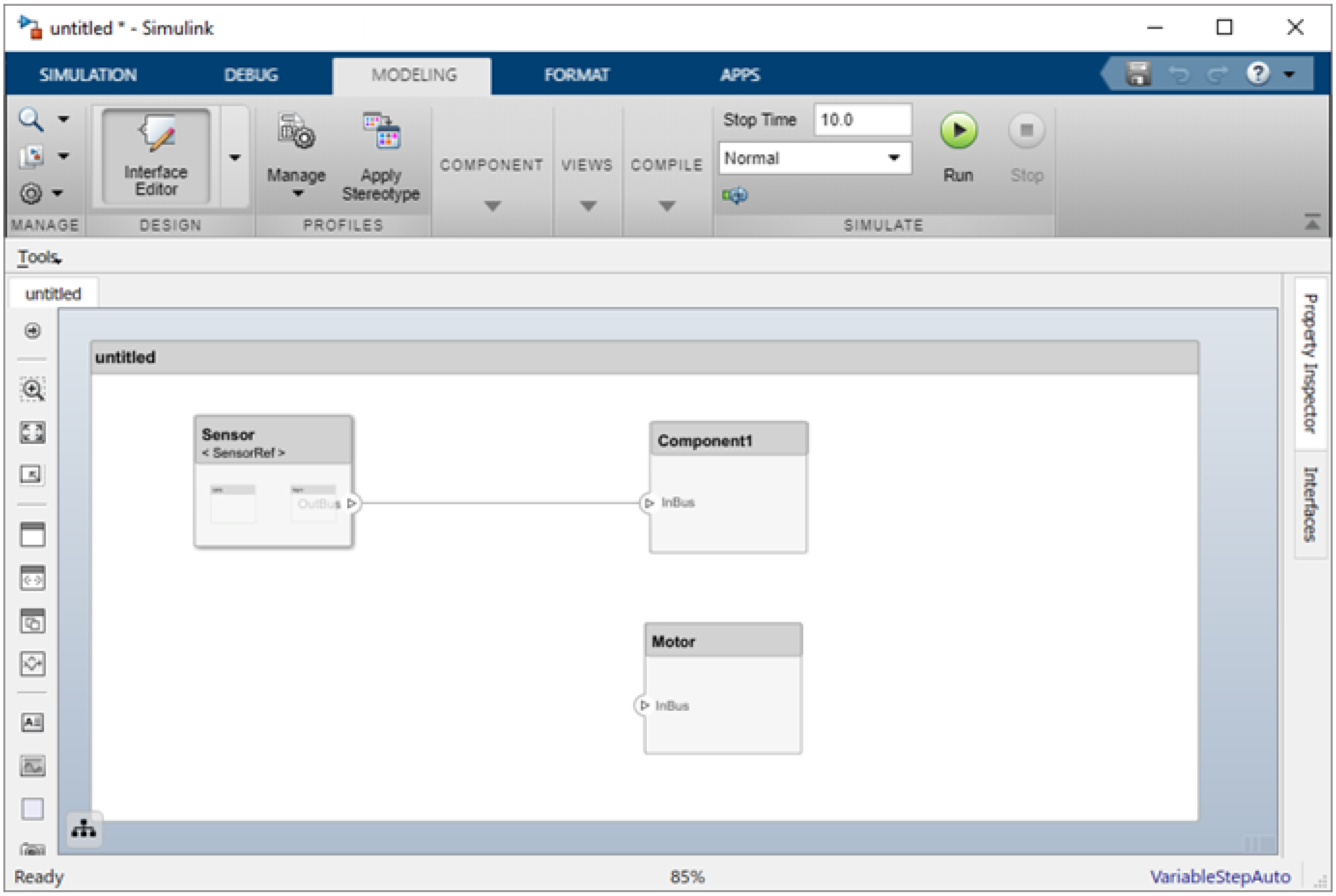
1 Right-click the component and select **Save as Architecture Model**.



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



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

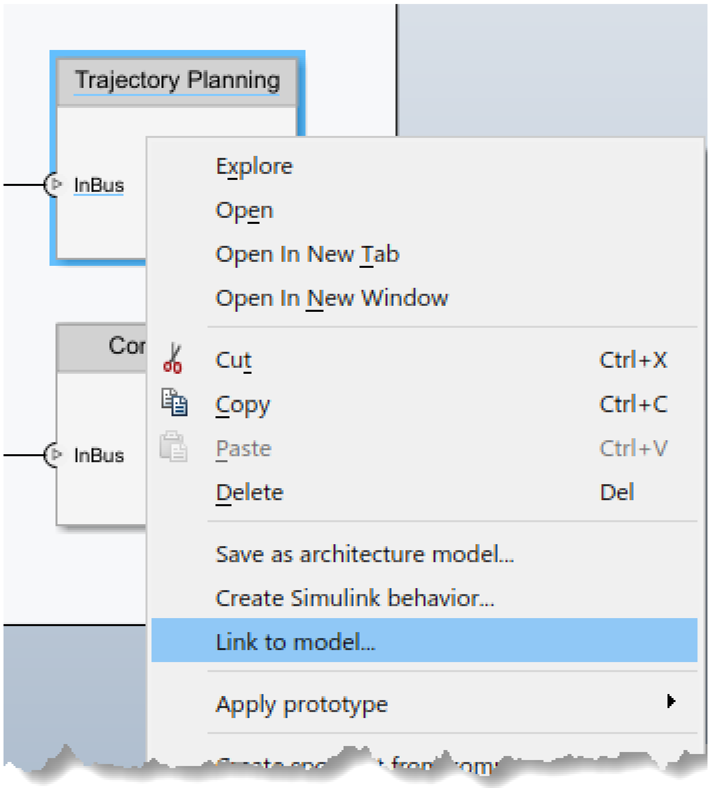


All architecture models can reference this new architecture model through linked components.

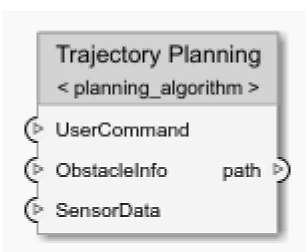
Use a Reference Architecture

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

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



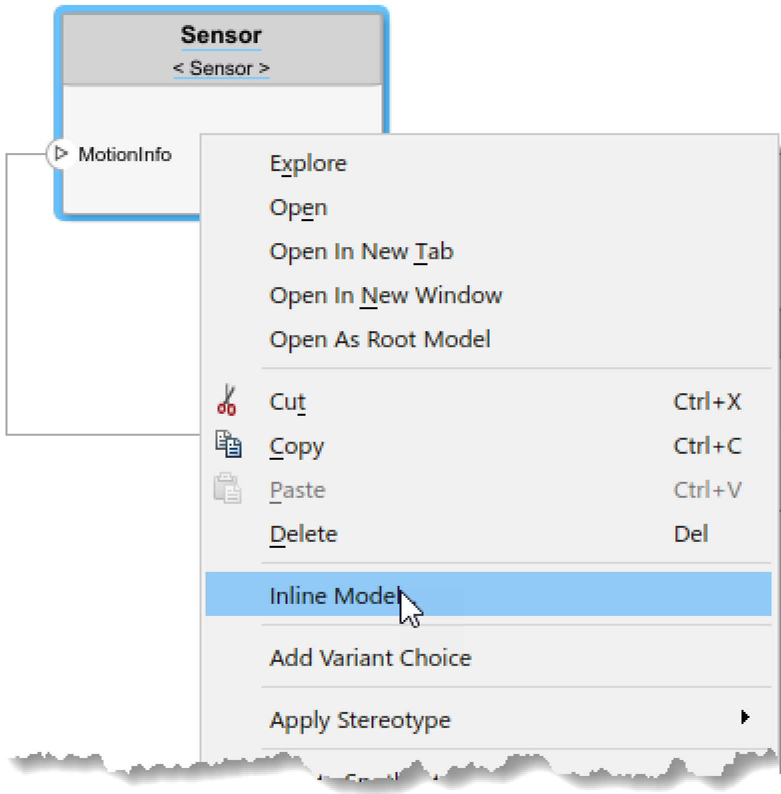
Provide the full path to the reference architecture. If the linked component has its own ports and components, this content is deleted during linking and replaced by that of the reference architecture. The ports of the linked component become the architecture ports in the reference architecture.



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

Inline a Reference Architecture

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



This operation provides two options:

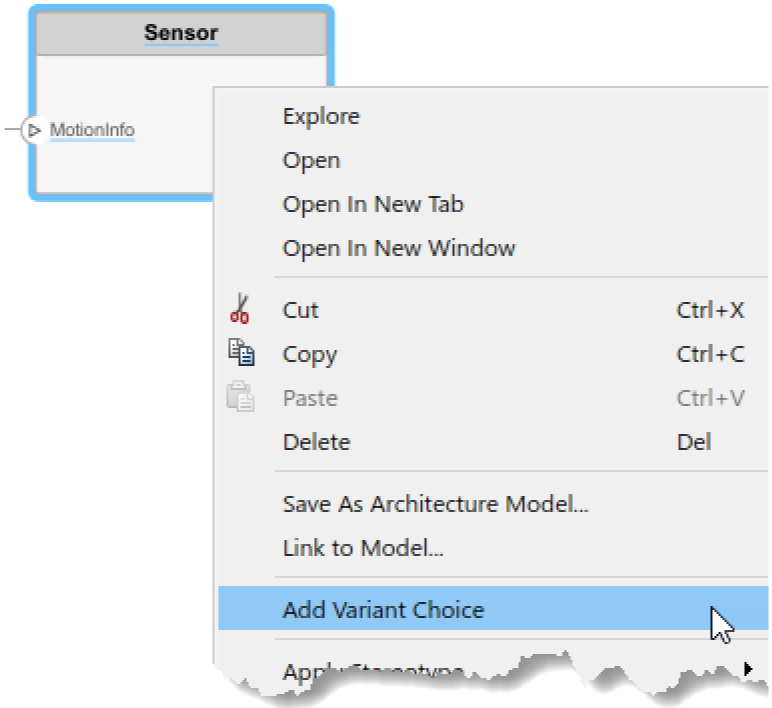
- Inline only interfaces — The ports and designated interfaces of the reference architecture are reflected on the component, but the composition is blank.
- Inline both interfaces and contents — Ports, interfaces, and subcomponents of the reference architecture are copied to the component.

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

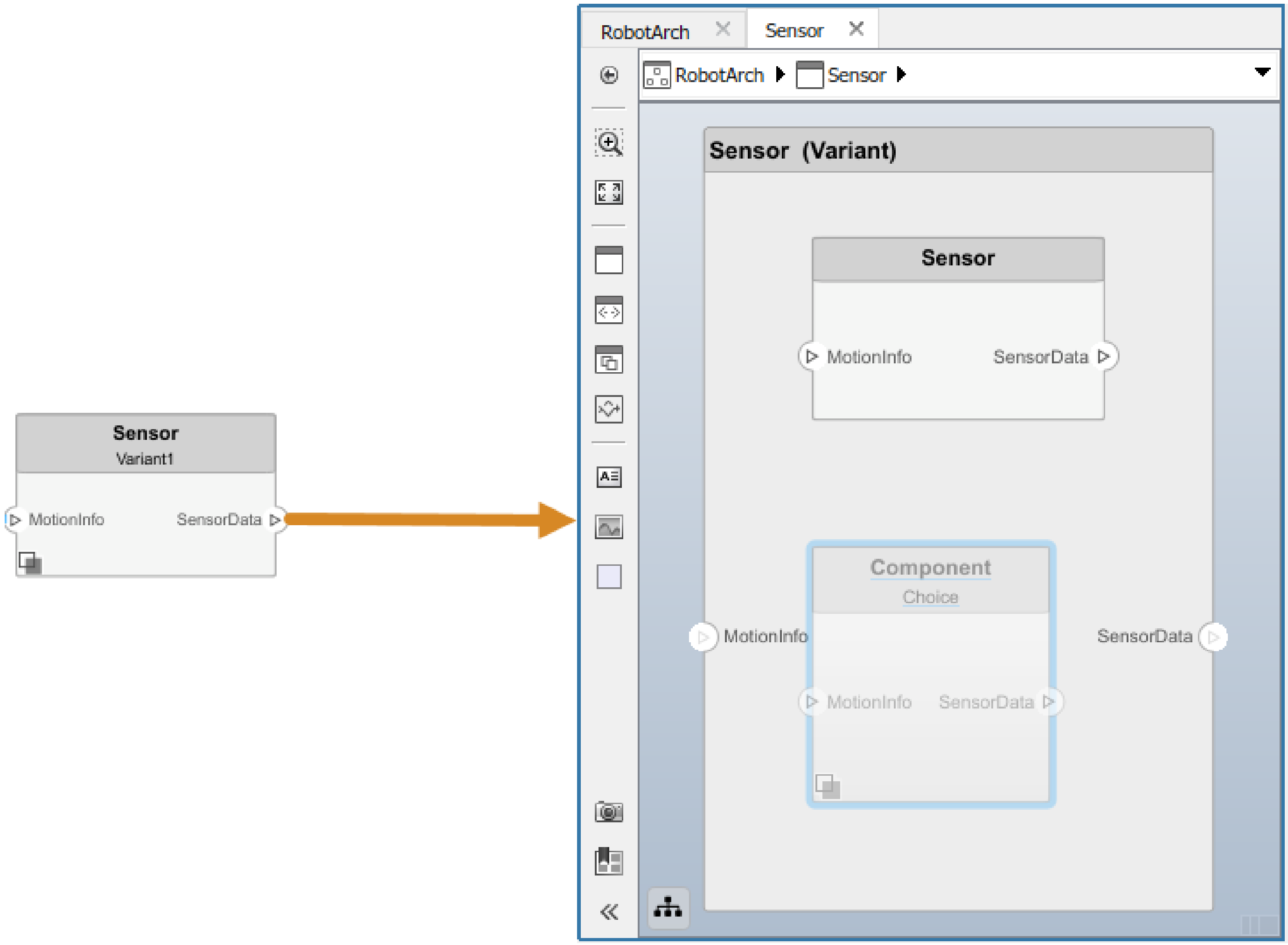
Create Variants

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

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



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

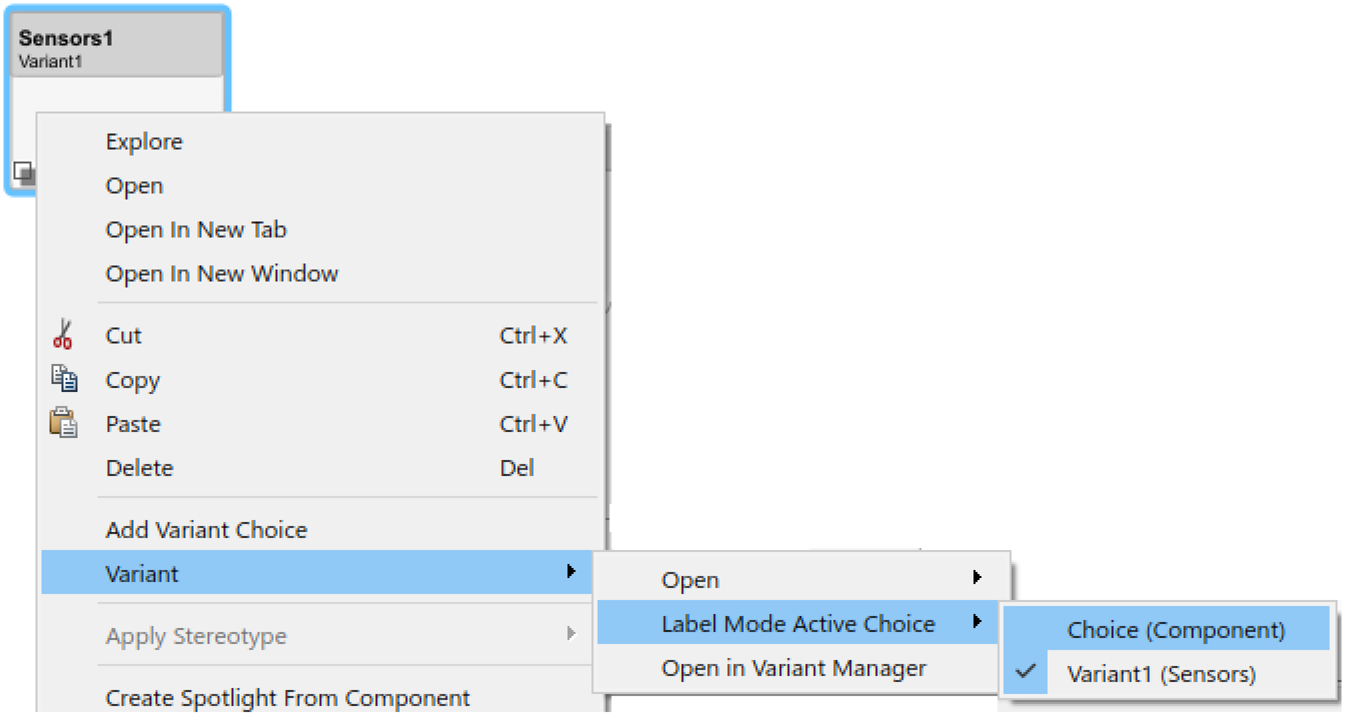


You can add more variant choices to a variant component using the **Add Variant Choice** option.

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

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

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



See Also

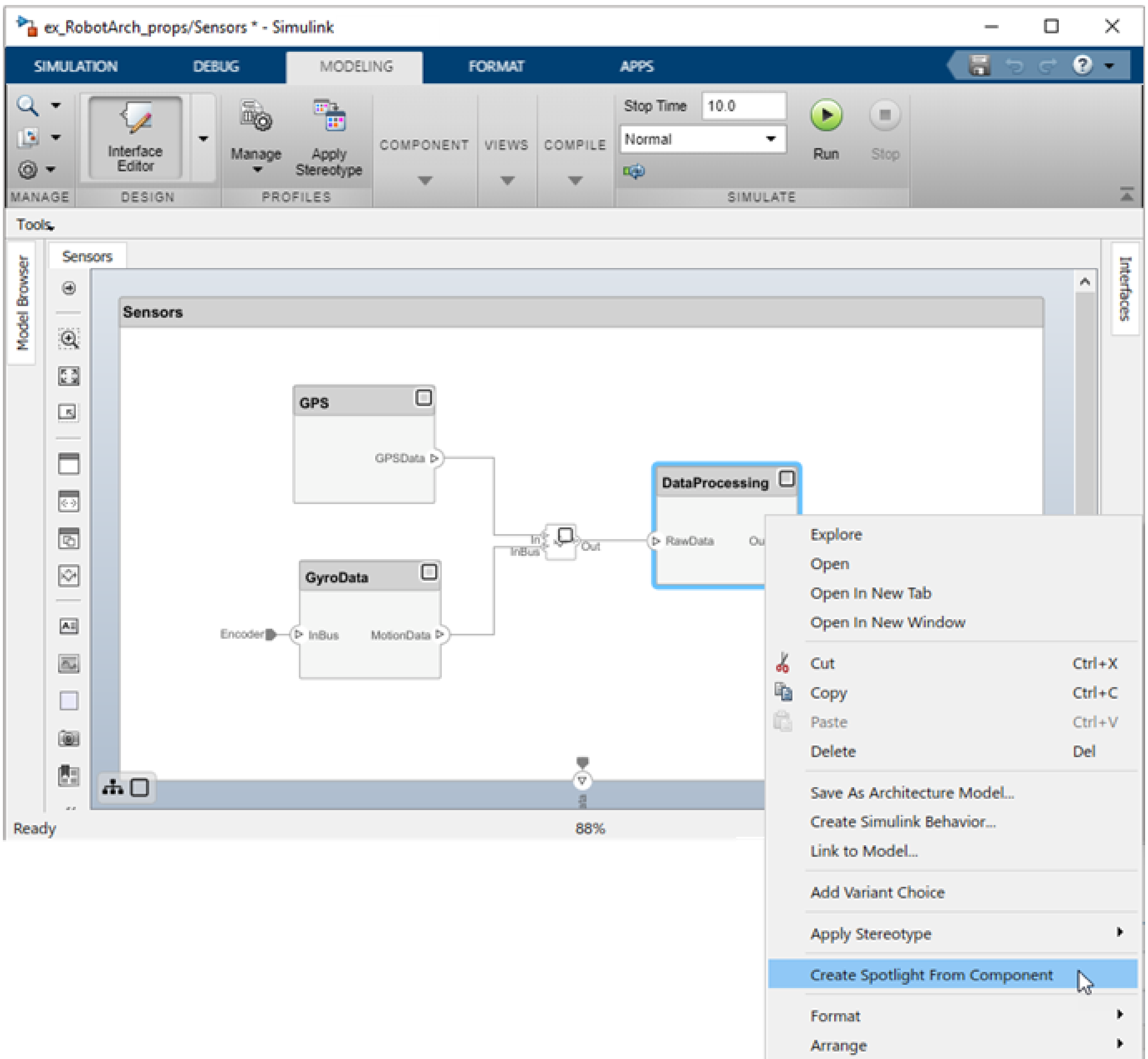
More About

- “Create a Simulink Behavior Model” on page 5-2
- “Link to an Existing Simulink Behavior Model” on page 5-4
- “Create Spotlight Views” on page 1-24

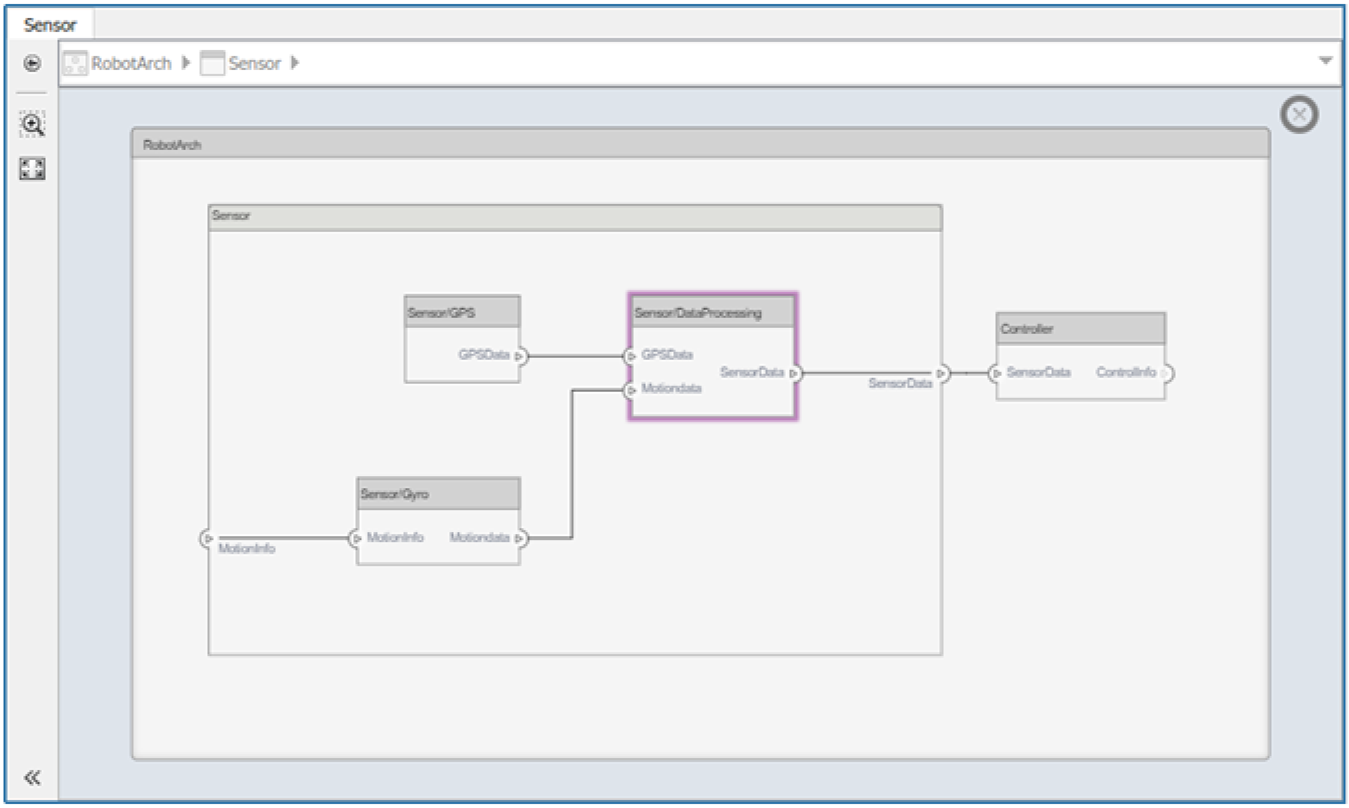
Create Spotlight Views

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

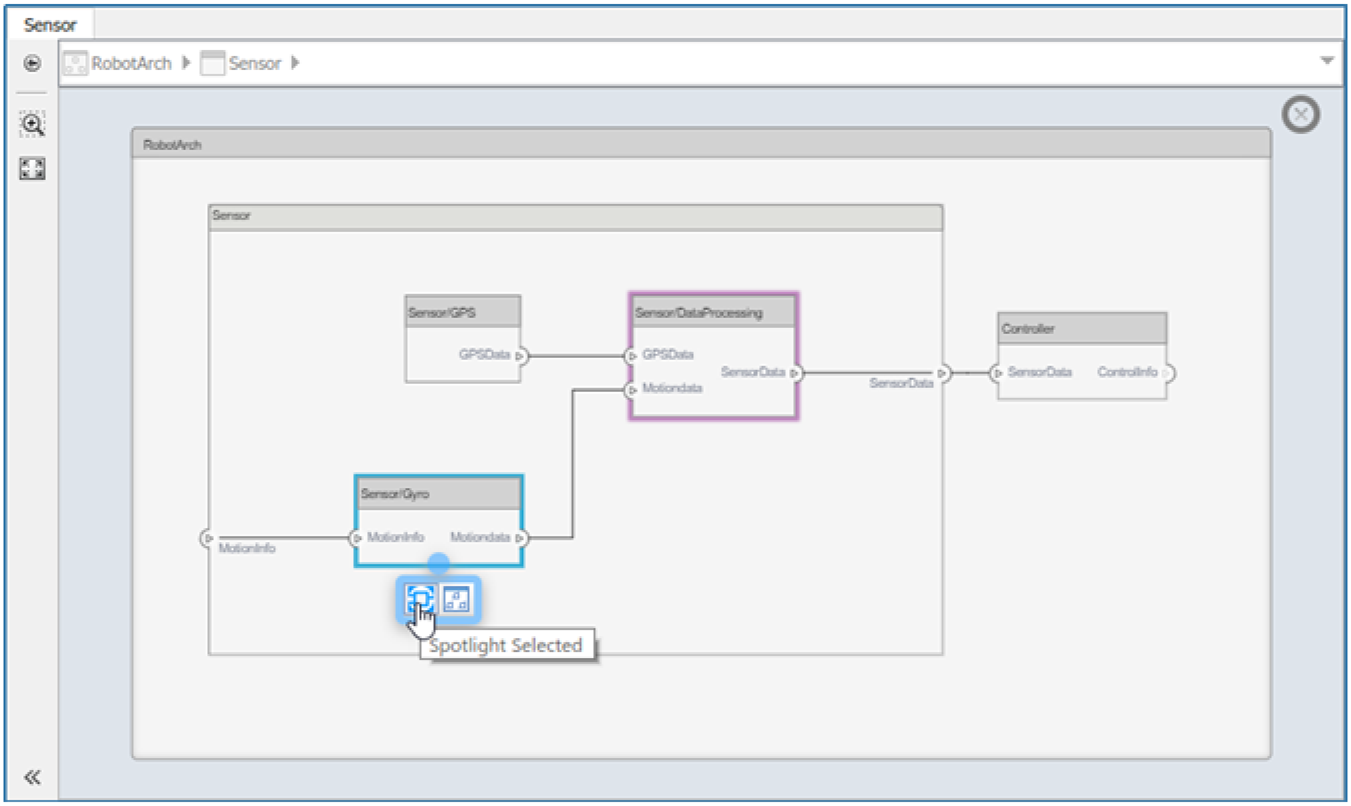
To create a spotlight from the composition, select the component of interest in the canvas, right-click and select **Create Spotlight from Component** either from the **Architecture** menu or the context menu.



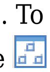
The spotlight view launches and shows all model elements to which the component connects in a transparent hierarchy. The spotlight diagram is laid out automatically and cannot be edited.



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



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

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

See Also

More About

- "Compose Architecture Visually" on page 1-2
- "Decompose and Reuse Components" on page 1-15

Build an Architecture Model from Command Line

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

Prepare Workspace

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

Build a Model

Add Components, Ports, and Connections

```
model = systemcomposer.createModel('mobileRobotAPI');
arch = model.Architecture;
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
planningPorts = addPort(components(2).Architecture,{'Command','SensorData','MotionCommand'},{'in
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
c_sensorData = connect(arch,components(1),components(2));
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Add and Connect an Architecture Port

Add a port on the architecture. This is an architecture port.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

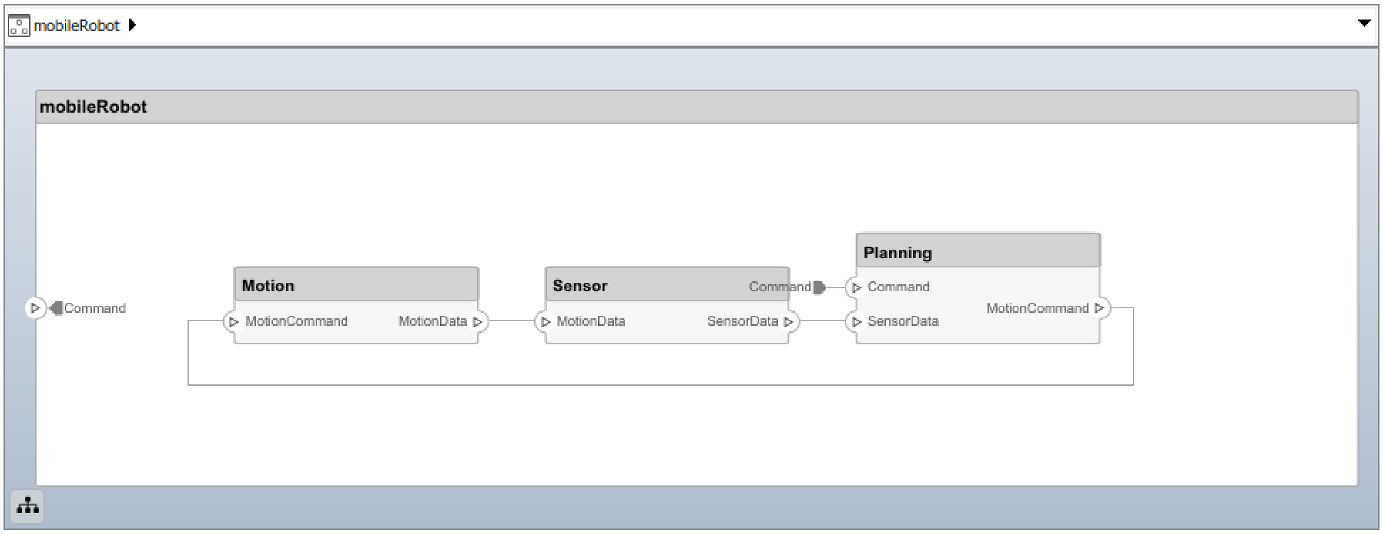
```
save(model)
```

Open the model

```
open_system(gcs);
```

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

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



Create and Apply Profile and Stereotypes

Profiles are `xml` files that can be applied to any model.

Create a Profile and Add Stereotypes

Create a profile.

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

Create a stereotype that applies to all element types:

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

Create stereotypes for different types of components. These types are dictated by design needs and are at the discretion of the user:

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

Create a stereotype for connections:

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

Add Properties

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

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

Apply Profile to Model

Apply profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'GeneralProfile.physicalComponent')
```

Apply the connector stereotype to all connections:

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

Apply the general element stereotype to all connectors and ports:

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

Set properties for each component:

```
setProperty(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'GeneralProfile.physicalComponent.Weight','2500');
```

Set the properties of connections to be identical:

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

Create an Interface

Create a data dictionary and add an interface:

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

Link the interface to the model:

```
linkDictionary(model,'SensorInterfaces.sldd');
```

Identify the interface in the dictionary:

```
interface_GPS = getInterface(model.InterfaceDictionary,'GPSInterface');
```

Set the interface for the port:

```
setInterface(sensorPorts(2),interface_GPS);
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Clean Up

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

```
% bdclose('mobileRobotAPI');
% systemcomposer.profile.Profile.closeAll;
% delete('SensorInterfaces.sldd');
```

Create Architecture Views Interactively

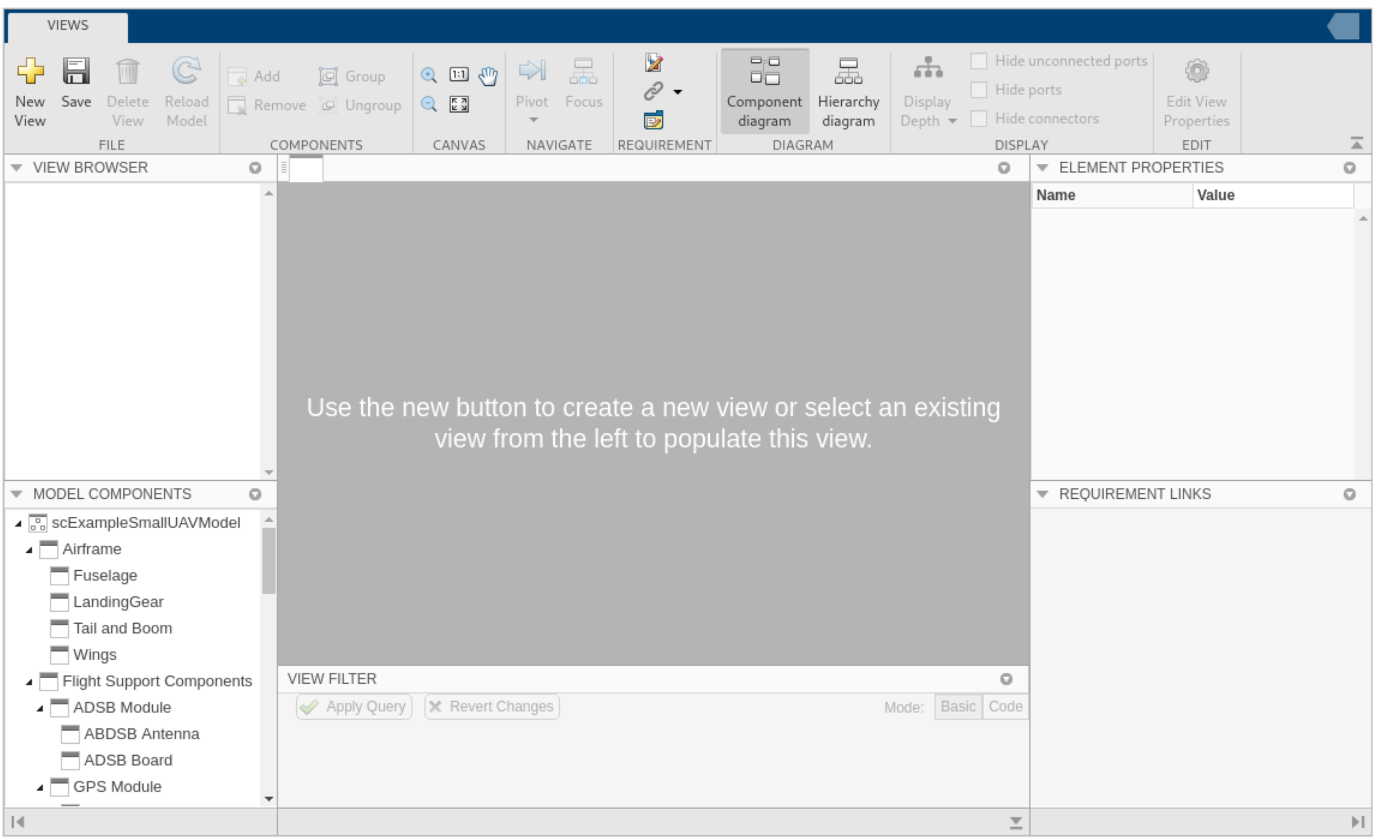
Typically, the structural hierarchy of a system differs from the hierarchy of the functional requirements. With architecture views, you can sketch the system based on different hierarchies. For example, you can author a system using the requirements. This allows you to better understand what components you need to satisfy your requirements while not necessarily focusing on the structure.

You can create an architecture view interactively. This example uses the architecture model for an unmanned aerial vehicle (UAV), `scExampleSmallUAV`, to create filtered and free form views. The view created shows the components having an interface for the light commands.

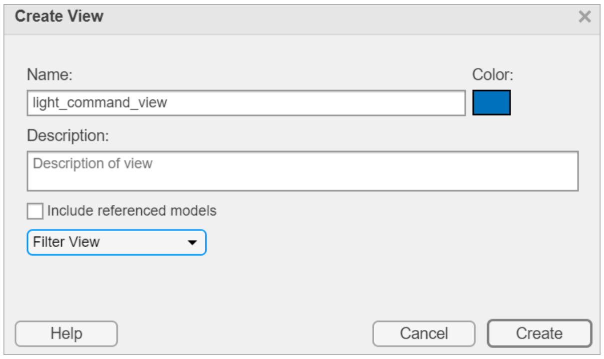
Create Filtered Views

To create a filtered view:

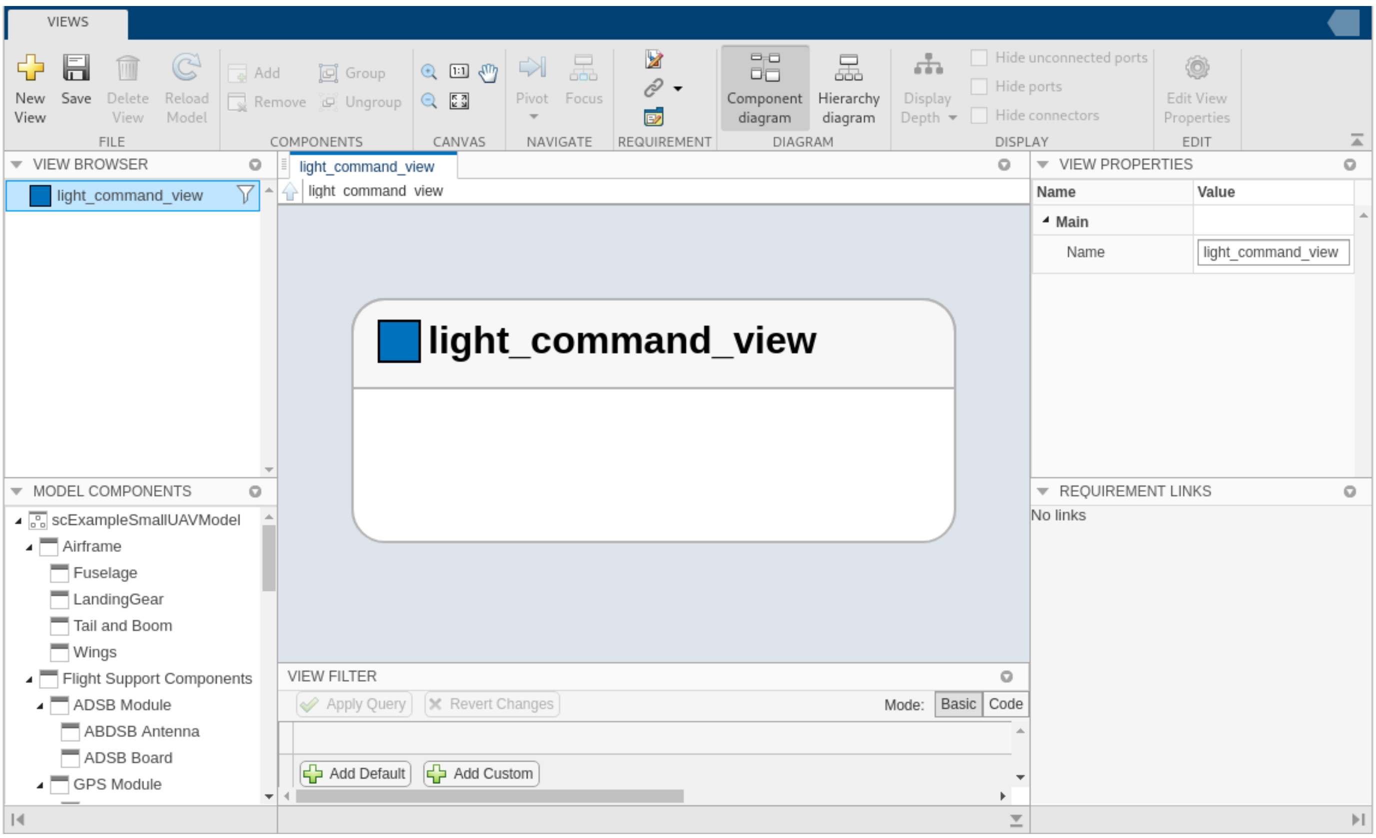
1. In the MATLAB command window, enter `scExampleSmallUAV`. The architecture model opens in the Simulink Editor.
2. In the **Views** section, click **Architecture Views** to open the Architecture Views Editor.



3. Click **New View** to open a Create View dialog box.
4. In the **Name** box, enter a name for this view. For example, `light_command_view`.

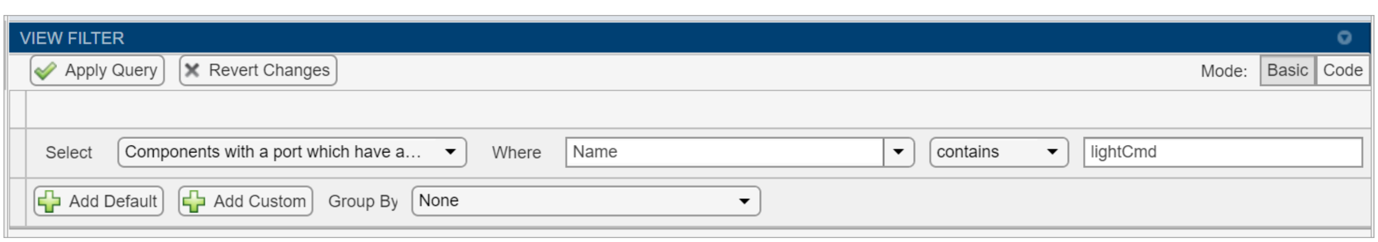


5 Select **Create** and observe that a new view is created.

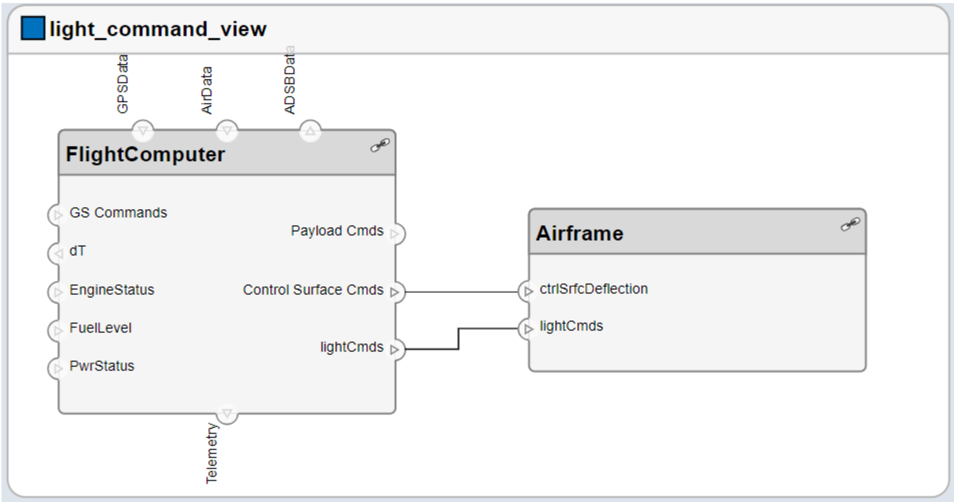


6 In the View Filter pane, select **Add Default** to add a new form-based criteria to the filter.

7 From the **Select** drop-down list, select `Components with a port which have an interface`. From the **Where** drop-down list, select `Name`, and in the text box, enter the name of an interface in the architecture model. For example, enter `lightCmd`.



8 Click **Apply Query**. The dialog box closes and an architecture view is created using the query from the **Filter** box. The view is filtered to select all the components for which the `lightCmd` interface is applied.

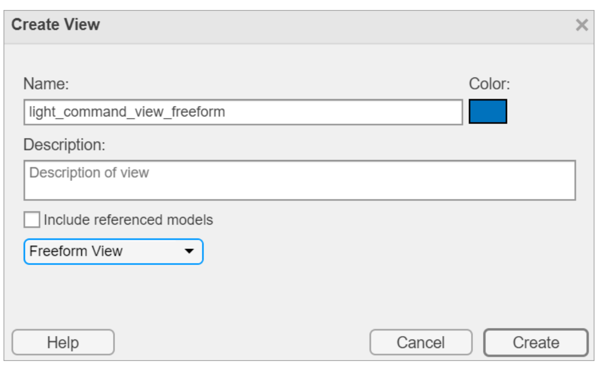


Create Freeform Views

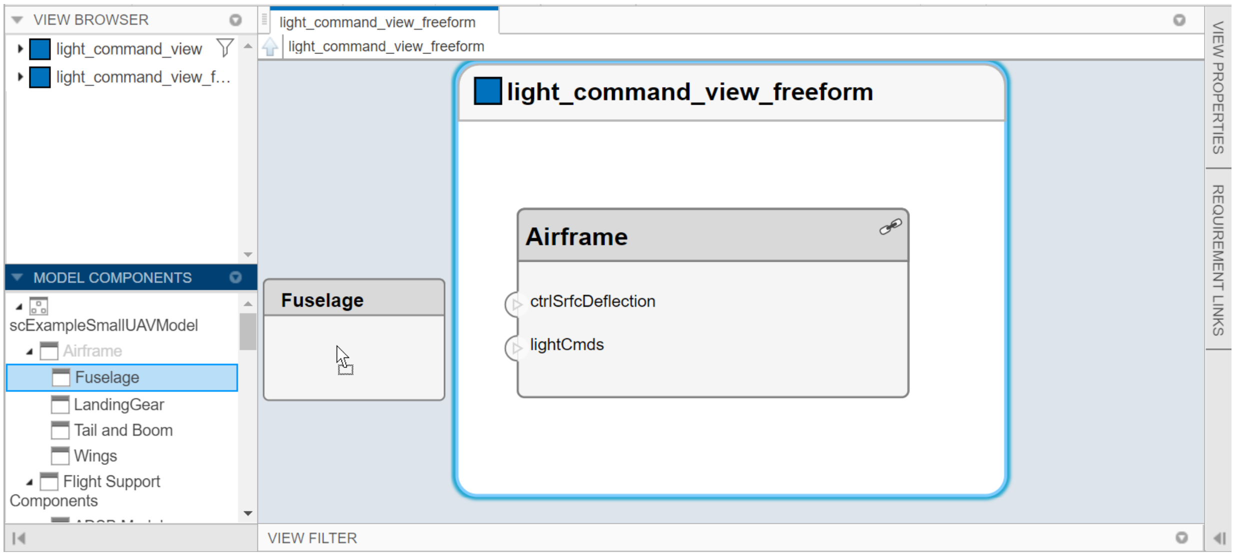
You can also create a freeform custom view without using a filter.

1 Click **New View**.

2 In the **Name** box, enter a name for this view. For example, use `light_command_view_freeform`. From the drop-down menu, select **Freeform View**. Select **Create**.

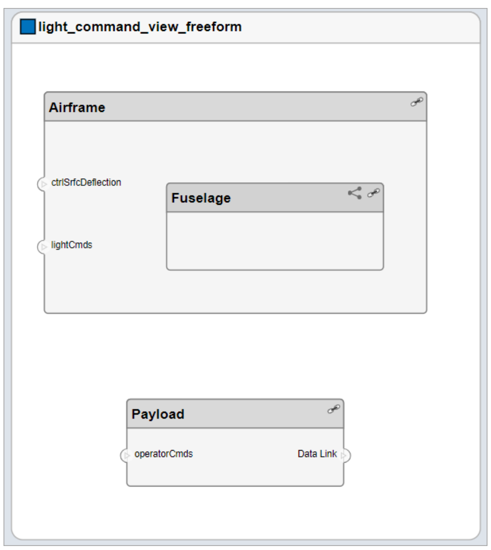


3 To add components to the view, drag and drop components from the Model Components. Drag and drop Airframe, Fuselage, and Payload components to your model. Alternatively, you can use the keyboard shortcut **Ctrl+I** to add component instantiations to your view.

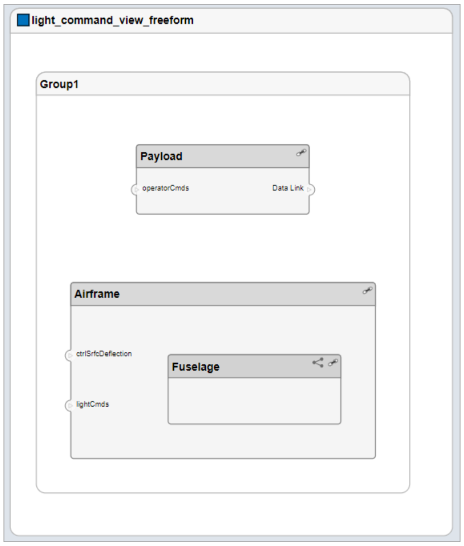


You can use the keyboard shortcut **Delete** to delete components from the view.

4 Observe that the free form view is created.

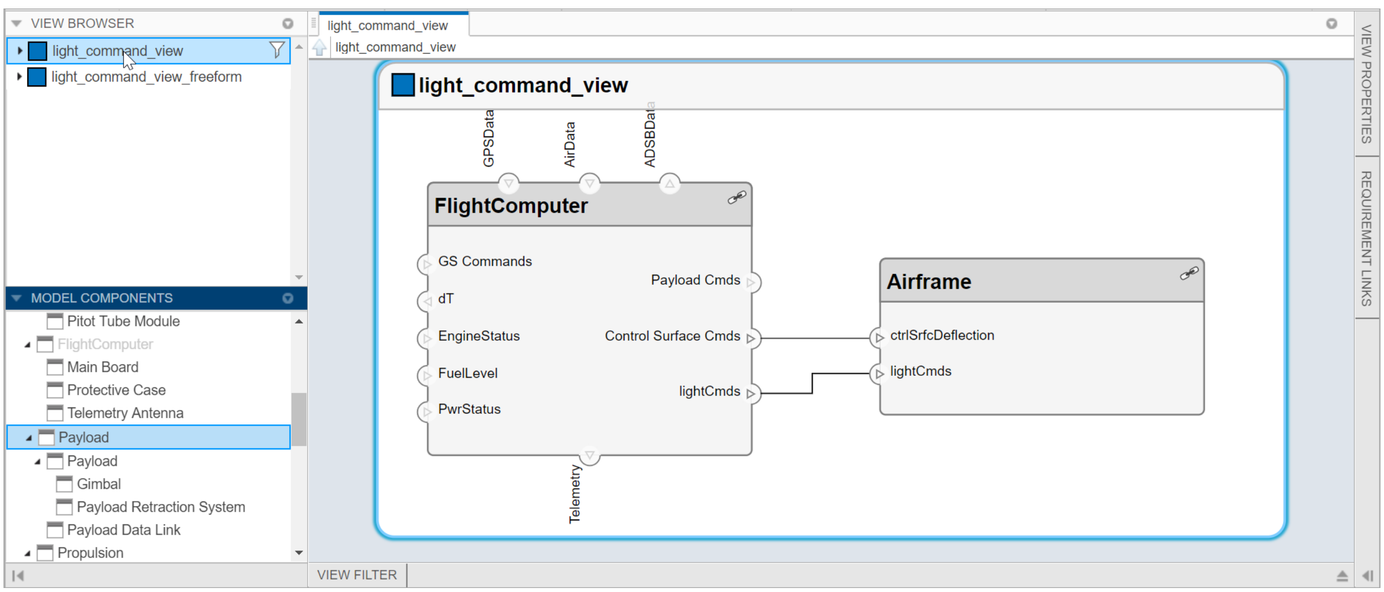


5 To group components, select (press **Shift** and click) the Airframe and Payload components and then the **Group**.



To ungroup components, select the components and click **Ungroup**.

6 Switch between the `light_command_view_freeform` and `light_command_view` by selecting the desired view from the View Browser.



Creating Architectural Views Programmatically

You can create an architecture view programmatically. This section constrains two examples for creating views programmatically from the MATLAB script `createArchitectureViews.m`.

1. Import the package where the queries are so you don't have to always use `systemcomposer.query`.

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

2. Open the Simulink project file.

```
scKeylessEntrySystem
```

3. Load the example model into Simulink.

```
zcModel = systemcomposer.loadModel('KeylessEntryArchitecture');
```

Example 1: Hardware Component Review Status

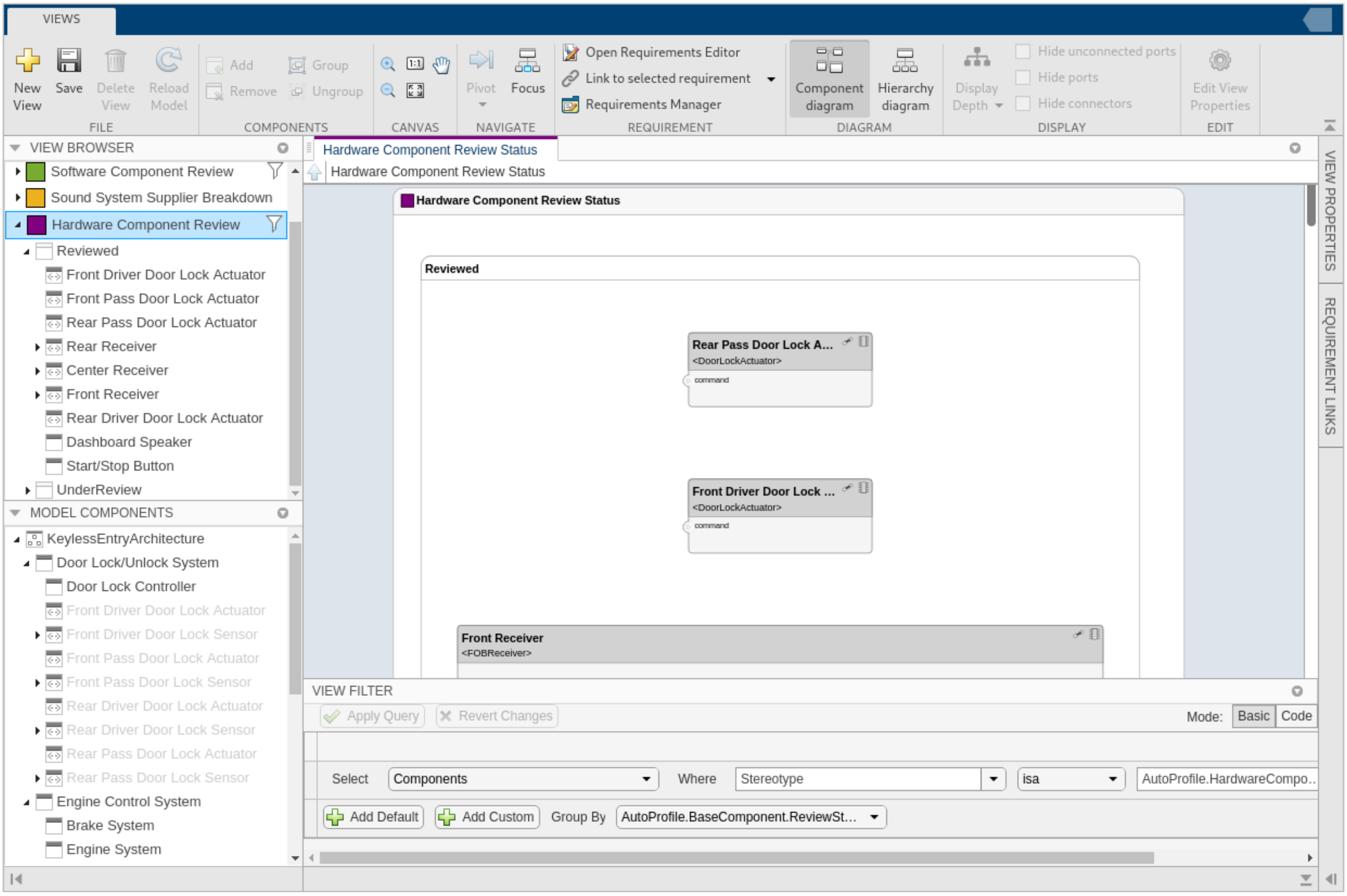
Create a filtered view that selects all of the hardware components in the architecture model and groups them using the `ReviewStatus` property.

1. Construct the query to select all of the hardware components.

```
hwCompQuery = HasStereotype(IsStereotypeDerivedFrom("AutoProfile.HardwareComponent"))

hwCompQuery = 

  HasStereotype with properties:

    AllowedParentConstraints: {[1×1 meta.class]}
               SubConstraint: [1×1 systemcomposer.query.IsStereotypeDerivedFrom]
              SkipValidation: 0
```

2. Use the query to create a view.

```
zcModel.createViewArchitecture("Hardware Component Review Status",...
    hwCompQuery,... % The query to use for the selection
    "AutoProfile.BaseComponent.ReviewStatus",... % The stereotype property to qualify by
    "IncludeReferenceModels",true,... % Include components in referenced models
    "Color","purple");

zcModel.openViews;
```



Example 2: FOB Locator System Supplier View

This example shows hot to create a freeform view that manually pulls the components from the FOB Locator system and then groups them using existing and new view components for the suppliers.

1 Create a view architecture.

```
fobSupplierView = zcModel.createViewArchitecture("FOB Locator System Supplier Breakdown",...
    "Color","lightblue");
```

2 Create a new view component for supplier D and add the FOB Locator module to it.

```
supplierD = fobSupplierView.createViewComponent("Supplier D");
supplierD.Architecture.addComponent("KeylessEntryArchitecture/FOB Locator System/FOB Locator Module");
```

3 Create a new view component for supplier A.

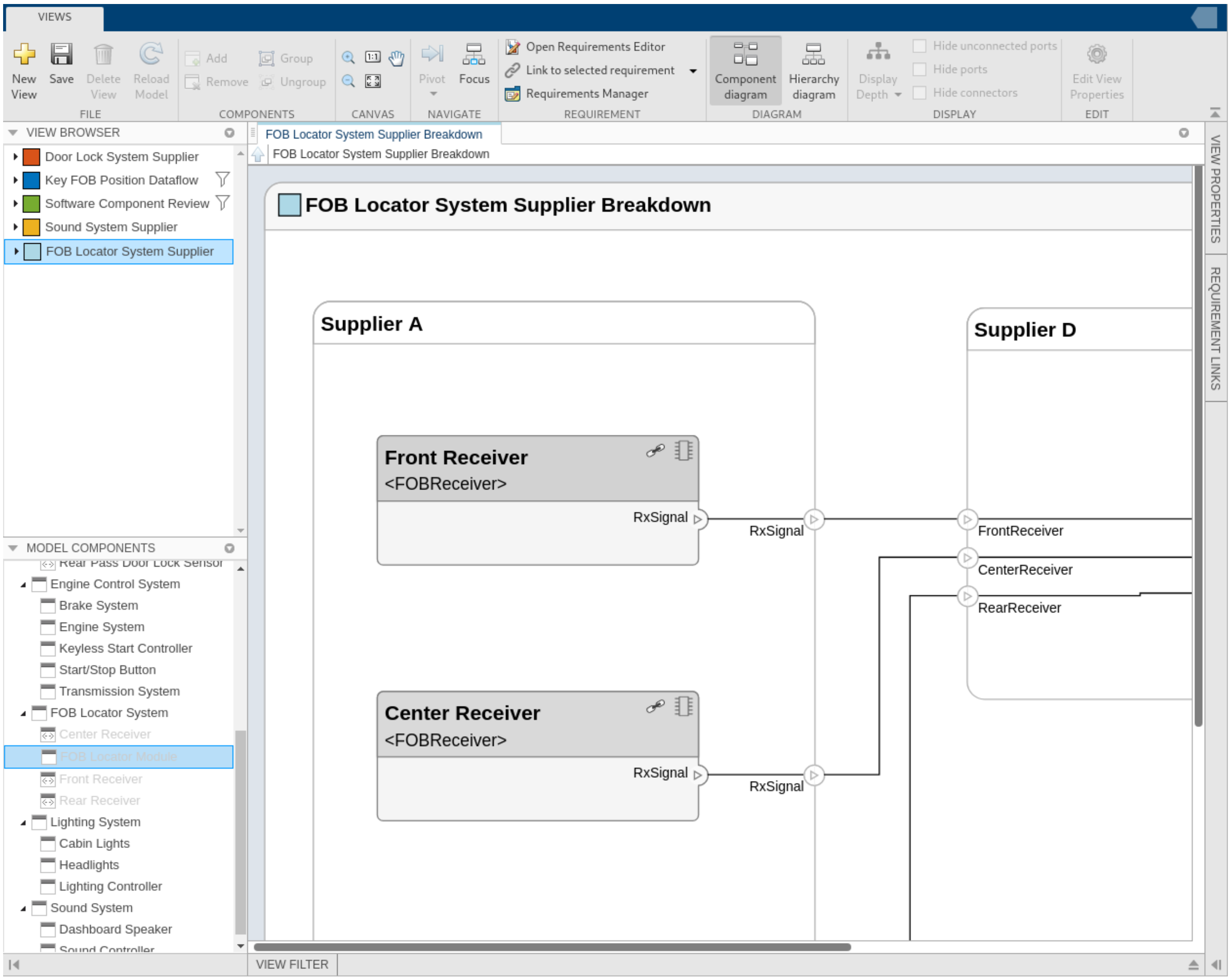
```
supplierA = fobSupplierView.createViewComponent("Supplier A");
```

4 Add each of the FOB receivers to view component.

```
FOBLocatorSystem = zcModel.lookup("Path", "KeylessEntryArchitecture/FOB Locator System");
receiverCompPaths = zcModel.find(...
    contains(systemcomposer.query.Property("Name"),"Receiver"),... % Find all the components which contain the name "Receiver"
    FOBLocatorSystem.Architecture);

for i = 1:numel(receiverCompPaths)
    % Add each of the components to supplier A
    supplierA.Architecture.addComponent(receiverCompPaths{i});
end
```

5 Open the Views Editor.

```
zcModel.openViews;
```



6 Close the model.

```
zcModel.close('Force');
```

Finding Elements in a System Composer Model Using Queries

This example shows how to find components in a system composer model using queries.

1 Open the MATLAB script.

```
open('scExampleModelFind')
```

2 Review the 6 example queries.

Import and Export Architecture Models

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

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

Define a Basic Architecture

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

- Components table
- Ports table
- Connections table

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

Components Table

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

- `Name` is component name.
- `ID` is a user-defined ID used to map child components and add ports to components.
- `ParentID` is 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 port name.
- `Direction` is an input or output port direction.
- `ID` is a user-defined port ID used to map ports to port connections.

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

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

Connections Table

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

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

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

Import a Basic Architecture

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

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

The basic architecture model opens.

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

Extend the Basic Architecture Import

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

Import Interfaces and Map Ports to Interfaces

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

Name	Parent	DataType	Dimensions	Units	Complexity	Minimum	Maximum
interface1							
elem1	interface1	interface3	1	""	real	"[]"	"[]"
interface2		1	1	""	real	"[]"	"[]"
elem2	interface1	1	1	""	real	"[]"	"[]"

Note Anonymous interfaces cannot be the data type of elements.

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

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

Import Variant 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.

Next example shows how to add a variant component `VarComp` with choices `Choice1` and `Choice2` and set `Choice2` as active choice.

Name	ID	ParentID	ReferenceModelName	ComponentType	ActiveChoice	VariantControl	VariantCondition	StereotypeName
root	0							
Component1	C1	0						
VarComp	V2	0		Variant	Choice2			
Choice1	C6	V2				petrol		
Choice2	C7	V2				diesel		
Component3	C3	0						
Component1_1	C4	C1						

Name	ID	ParentID	ReferenceModelName	ComponentType	ActiveChoice	VariantControl	VariantCondition	StereotypeName
Component1_2	C5	C1						

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

Apply Stereotypes and Set Property Values on Imported Model

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

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

Name	ID	ParentID	StereotypeNames	UAVComponent_OnboardElement_Mass	AVComponent_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:

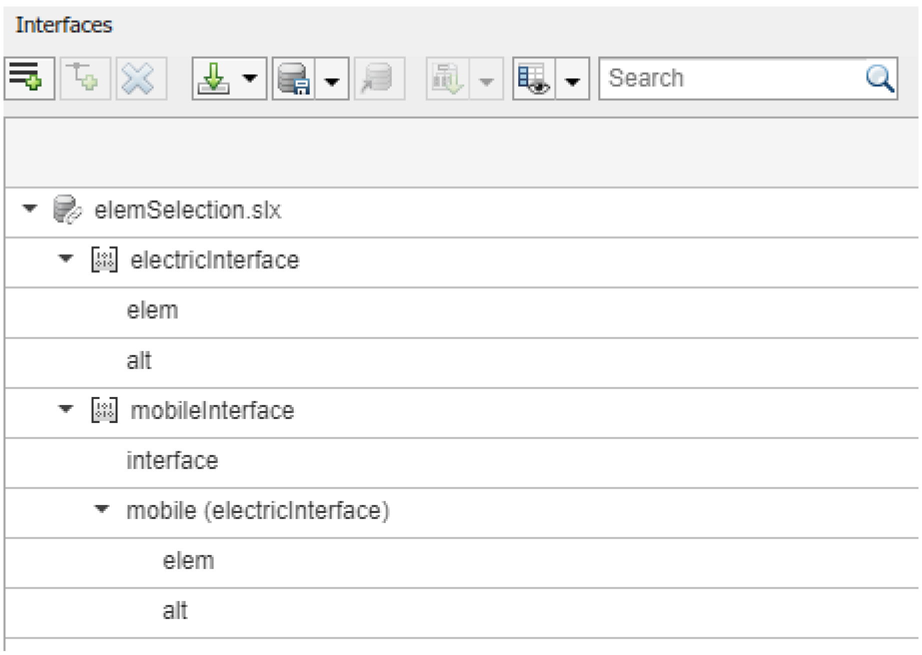
- `Label` is the name of the requirement.
- `SourceID` is the architecture element to which the requirement is attached.
- `DestinationType` is how requirements are saved.
- `DestinationID` is where the requirement is located.
- `Type` is the requirement type.

Label	SourceID	DestinationType	DestinationID	Type
rset#1	components:1	linktype_rmi_slreq	C:\Temp\rset.slreqx#1	Implement

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

Specify Multiple Elements on an Architecture Port

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



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

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

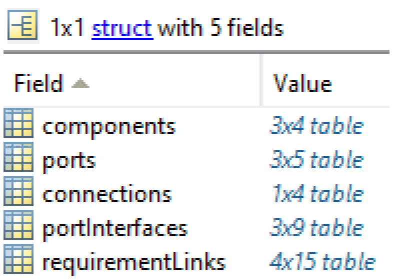
RootToRoot	5	5	6	mobile,mobile.alt	

Export an Architecture

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

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

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



See Also

`systemcomposer.exportModel` | `systemcomposer.importModel`

Display Component Hierarchy Using Hierarchy Views

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

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

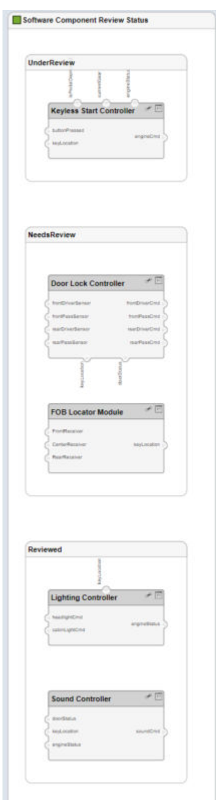
This example uses an architecture model representing a keyless entry system for a vehicle to show the Hierarchy View. For more information about the keyless entry system, see “Modeling System Architecture of Keyless Entry System” on page 6-30.

Switch Between Component Diagram and Hierarchy Diagram

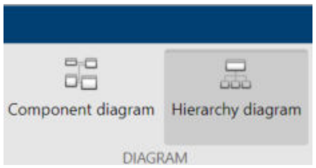
1. To open the `scKeylessEntrySystem` project, use the command below.

   ```
   scKeylessEntrySystem
   ```

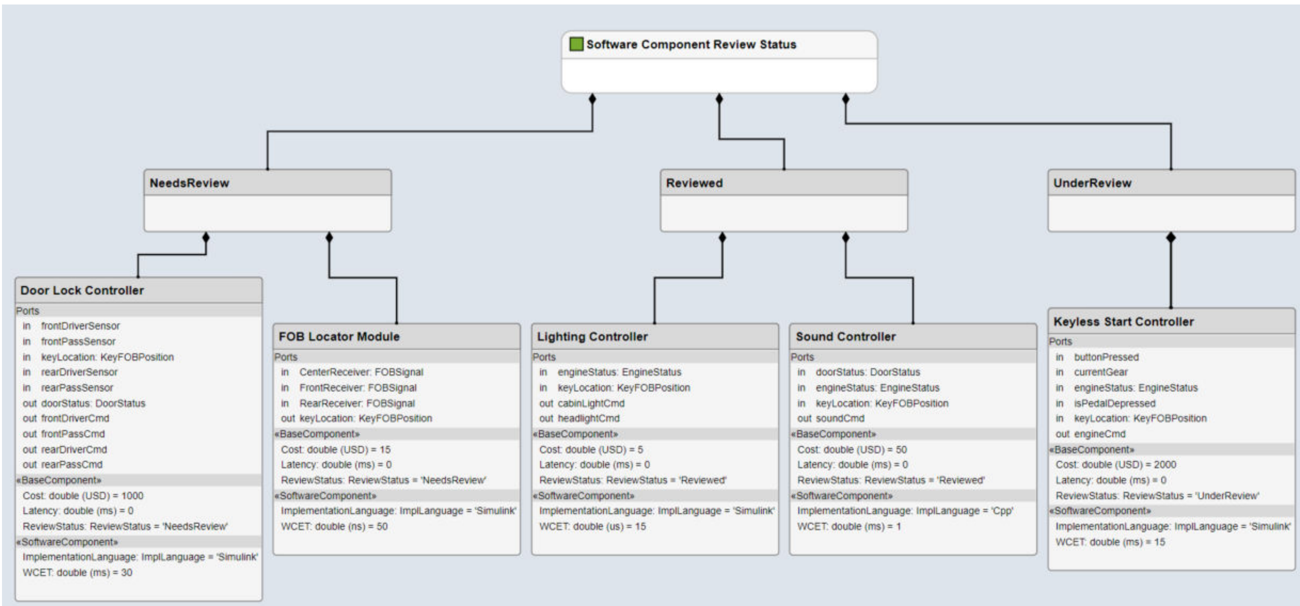
2. To open the architecture views, on the **Modeling** tab, select **Architecture Views**.
3. From the View Browser, select **Software Component Review** to display the component diagram.

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



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



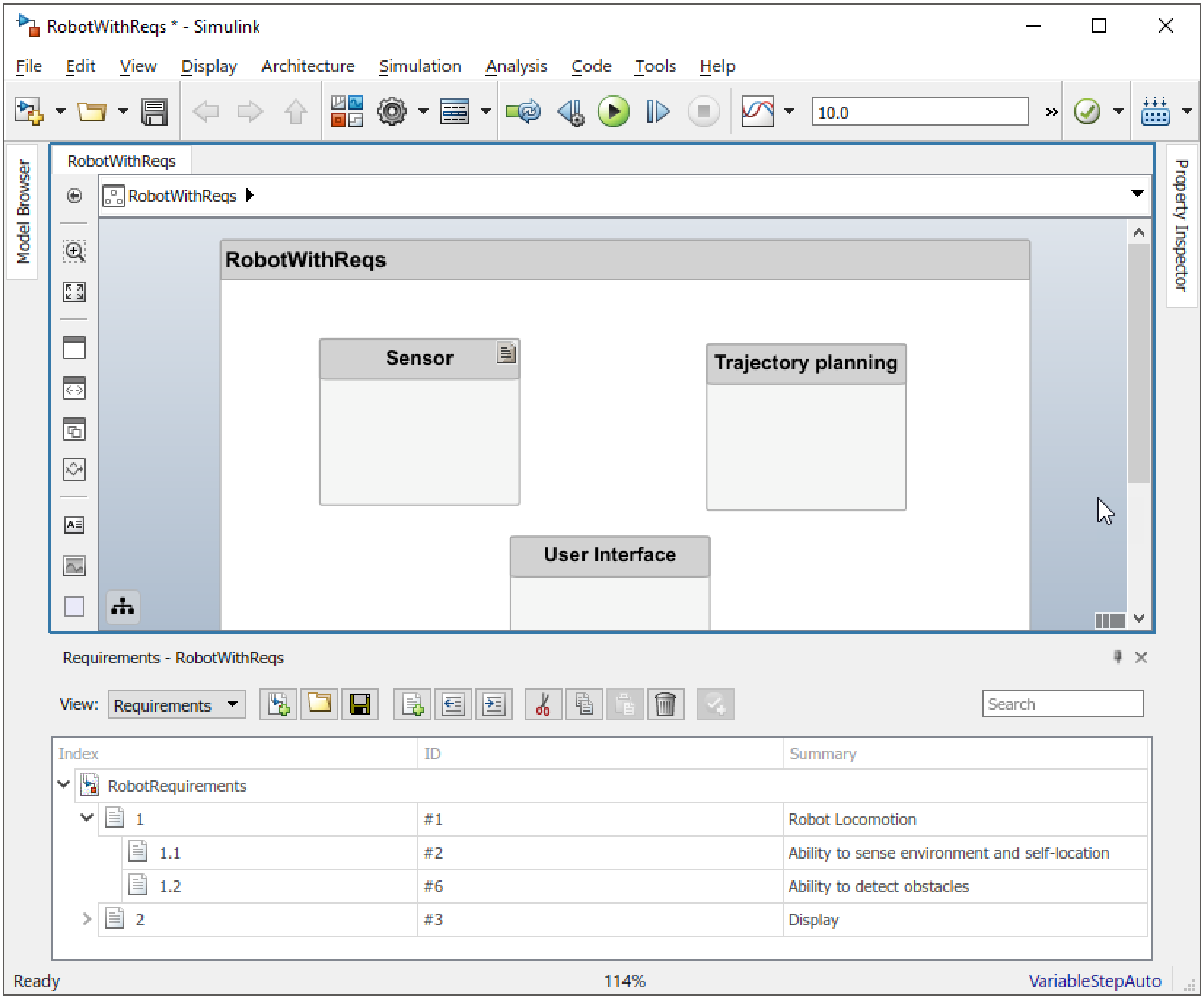
The single root of the hierarchy diagrams show a single root, which is the view specification itself. The root corresponds to the containing system box shown in the component diagram. The connections in the hierarchy diagram originate from the child components and end with a diamond symbol at the parent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

2

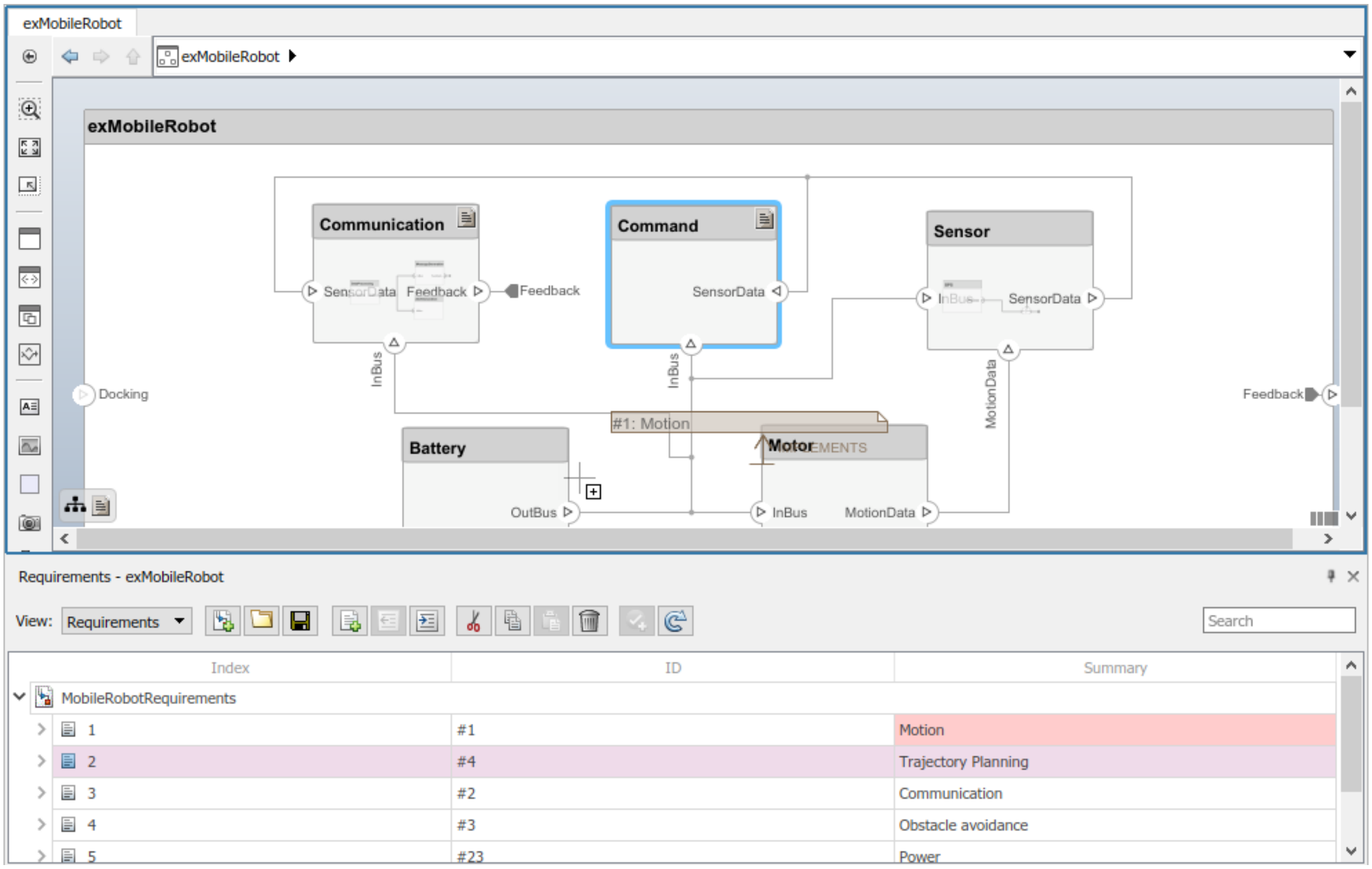
Requirements

Manage Requirements

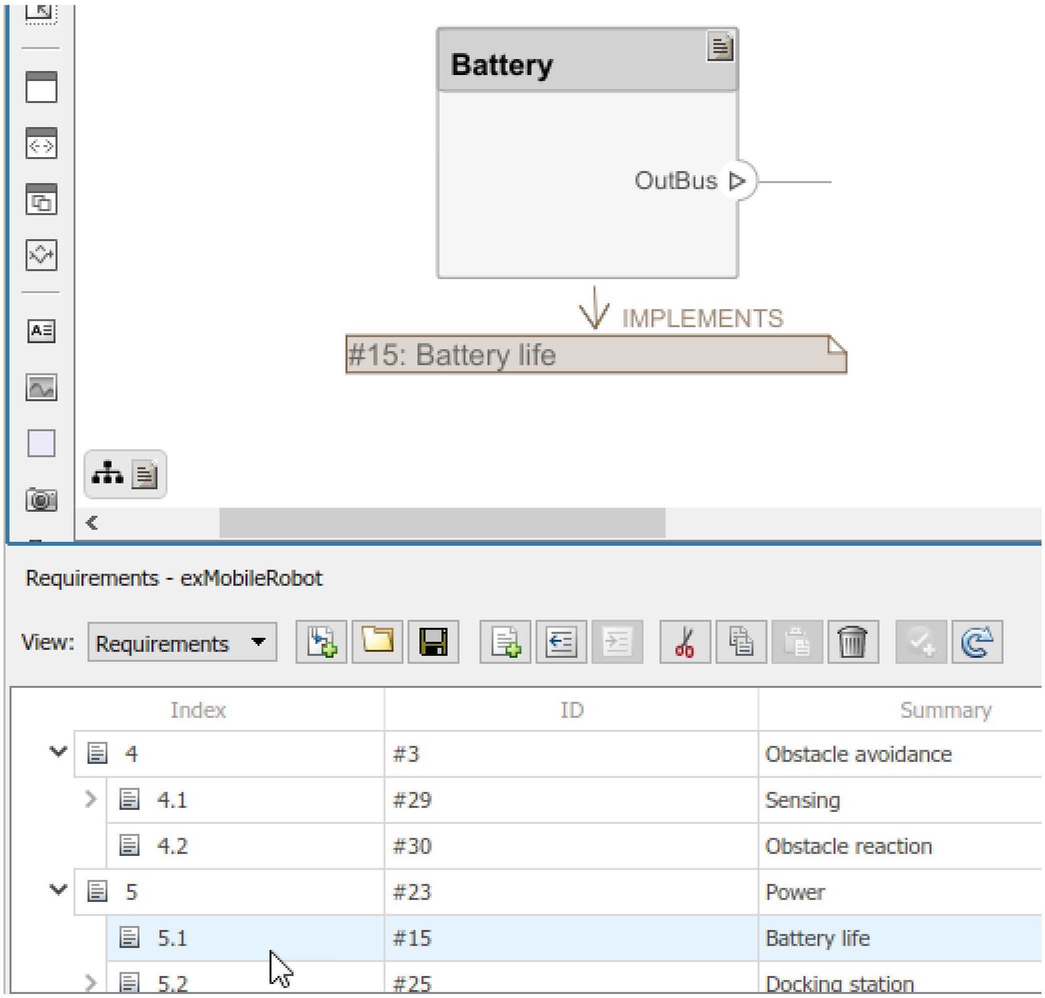
Manage requirements and architecture model together in the **Requirements** perspective from Simulink Requirements™. Select **Apps > Requirements Manager**.



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

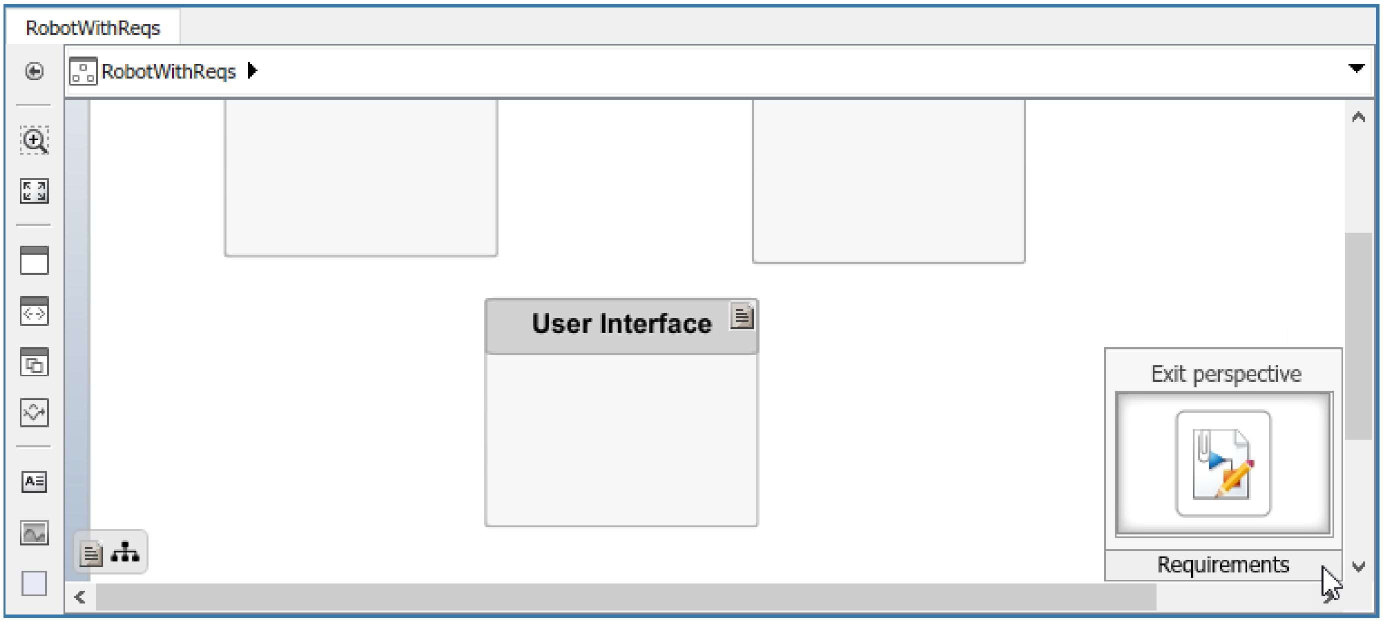


To directly create a link, drag a requirement onto a component.



You can close the annotation that shows the link as necessary. This 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, see “Manage Navigation Backlinks in External Requirements Documents” (Simulink Requirements).

See Also

More About

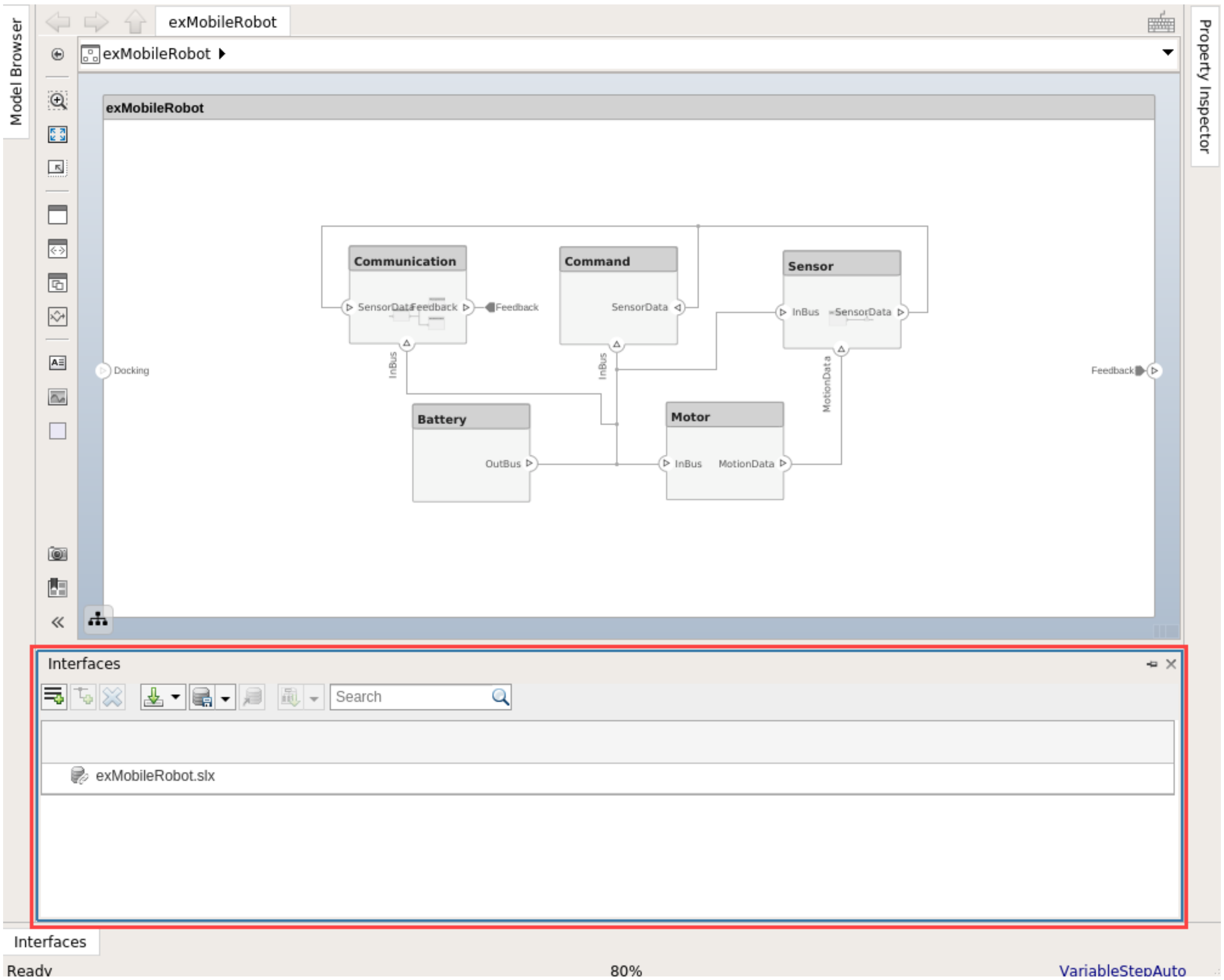
- “Link Blocks and Requirements” (Simulink Requirements)

Interface Management

Define Interfaces

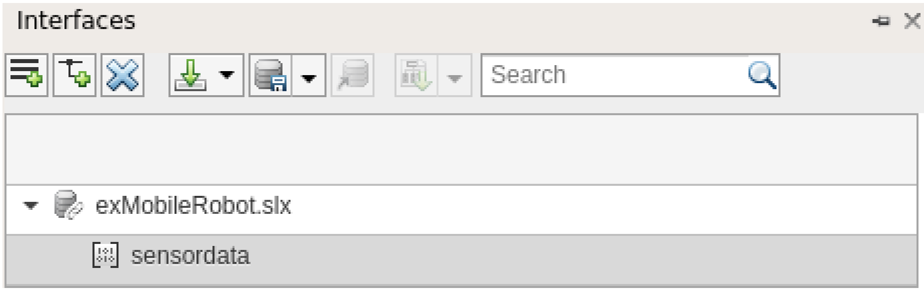
A system engineering solution includes a formal definition of the interfaces between components. A connection shows that two components have an output-to-input relationship; an interface defines the type, dimensions, units, and structure of the data. You can define interfaces using the Interface Editor.

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

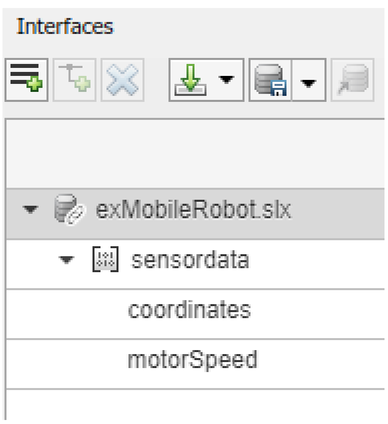


Create Interface

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

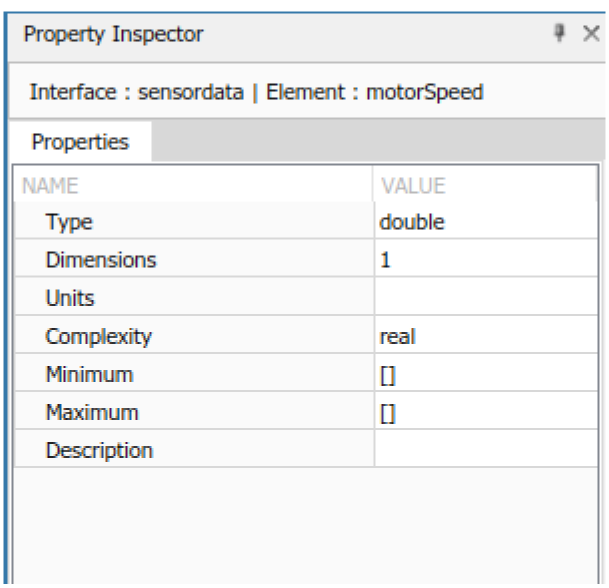


To add an element to the interface, click the  icon. Interface and element names must be valid variable names.



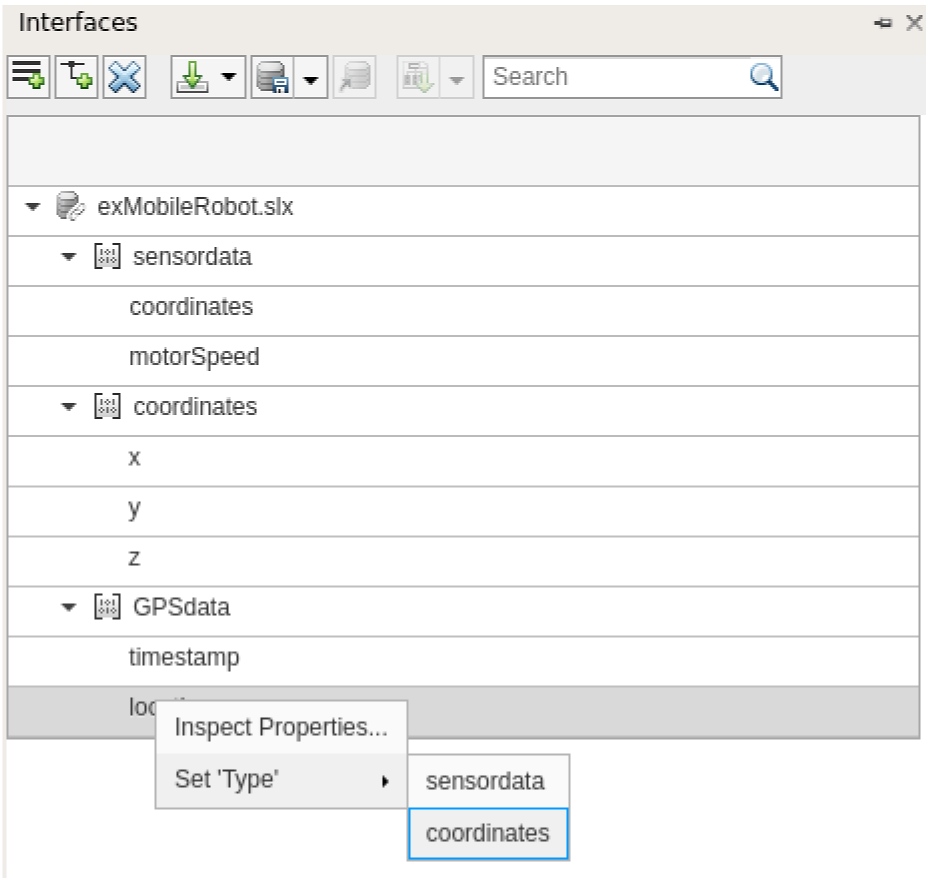
You can delete interfaces and elements in the Interface Editor using the  button.

You can view and edit the properties of an element in the Property Inspector. Right-click the interface element and select **Inspect Properties**.



A hierarchical interface contains another interface. Create a hierarchical interface by assigning an interface as the type of an interface element.

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



The defined interfaces become part of the model data dictionary.

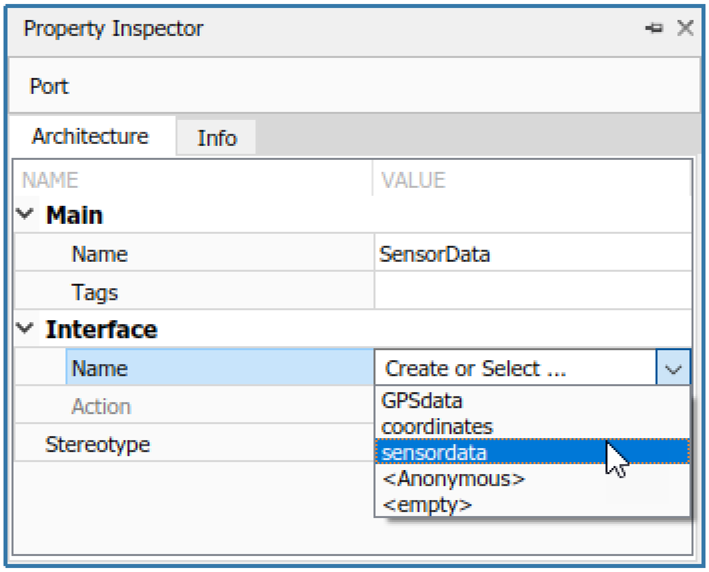
See Also

More About

- "Assign Interfaces to Ports" on page 3-5
- "Save, Link, and Delete Interfaces" on page 3-8

Assign Interfaces to Ports

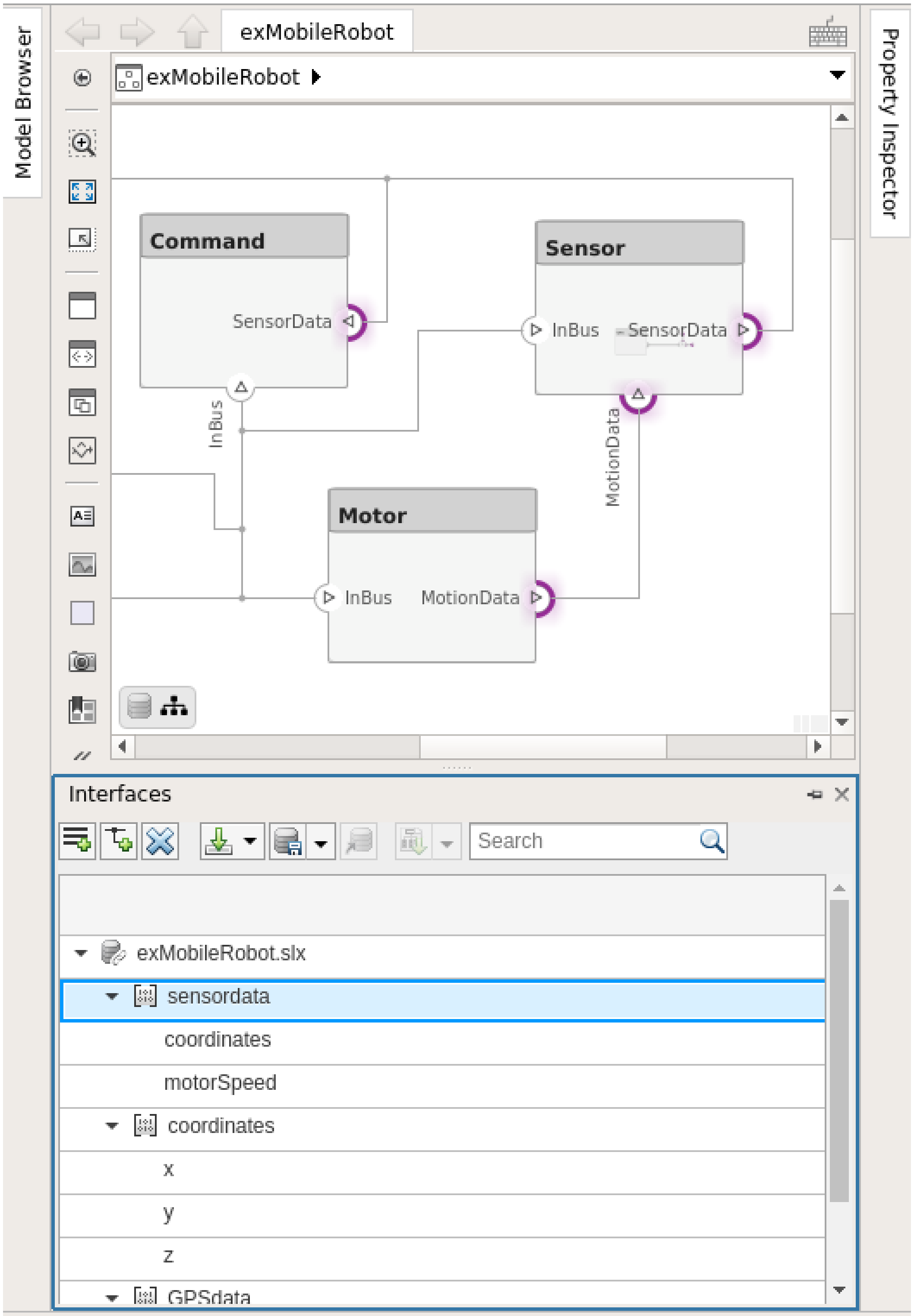
Associate a port with an interface using the Property Inspector. To open the Property Inspector, locate it in the toolstrip in the **Design** section drop down. To show the **SensorData** port properties, highlight the port in the model. Expand **Interface**, and select the `sensordata` interface in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



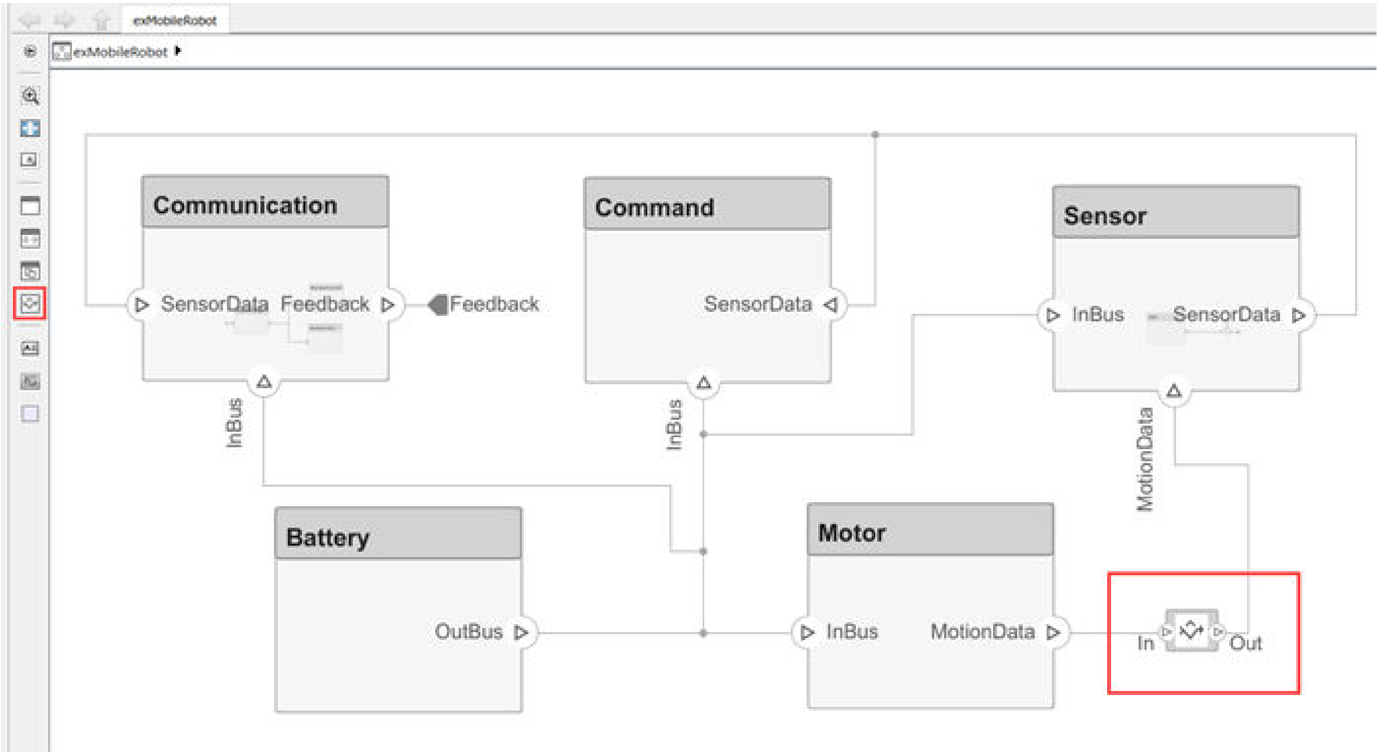
You can select an interface in the model data dictionary (see “Define Interfaces” on page 3-2), or create an anonymous interface — an interface of unstructured data whose properties are valid for that port only. An anonymous interface does not have a structure, but does have prescribed properties such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

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

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



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



Change the number of input ports on an Adapter block the same way you add and remove component ports. For more information, see “Ports” on page 1-8.

See Also

More About

- “Define Interfaces” on page 3-2
- “Save, Link, and Delete Interfaces” on page 3-8
- “Interface Adapter” on page 3-14

Save, Link, and Delete Interfaces

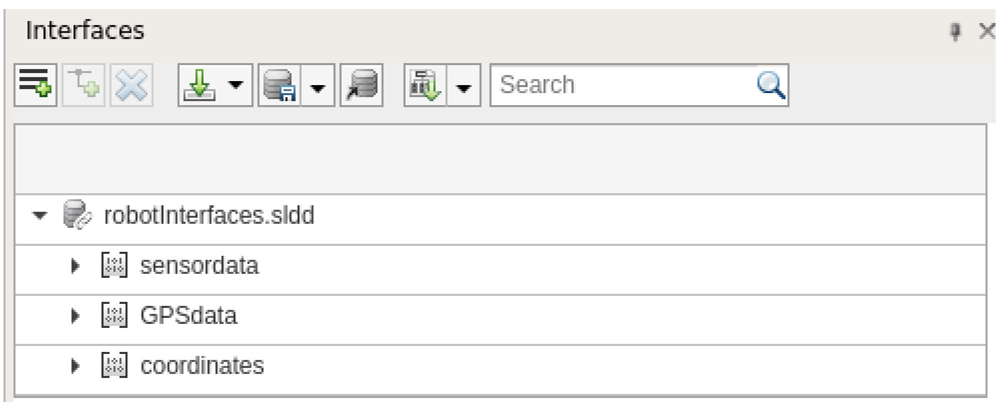
In this section...

Store Interfaces in a Data Dictionary

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

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

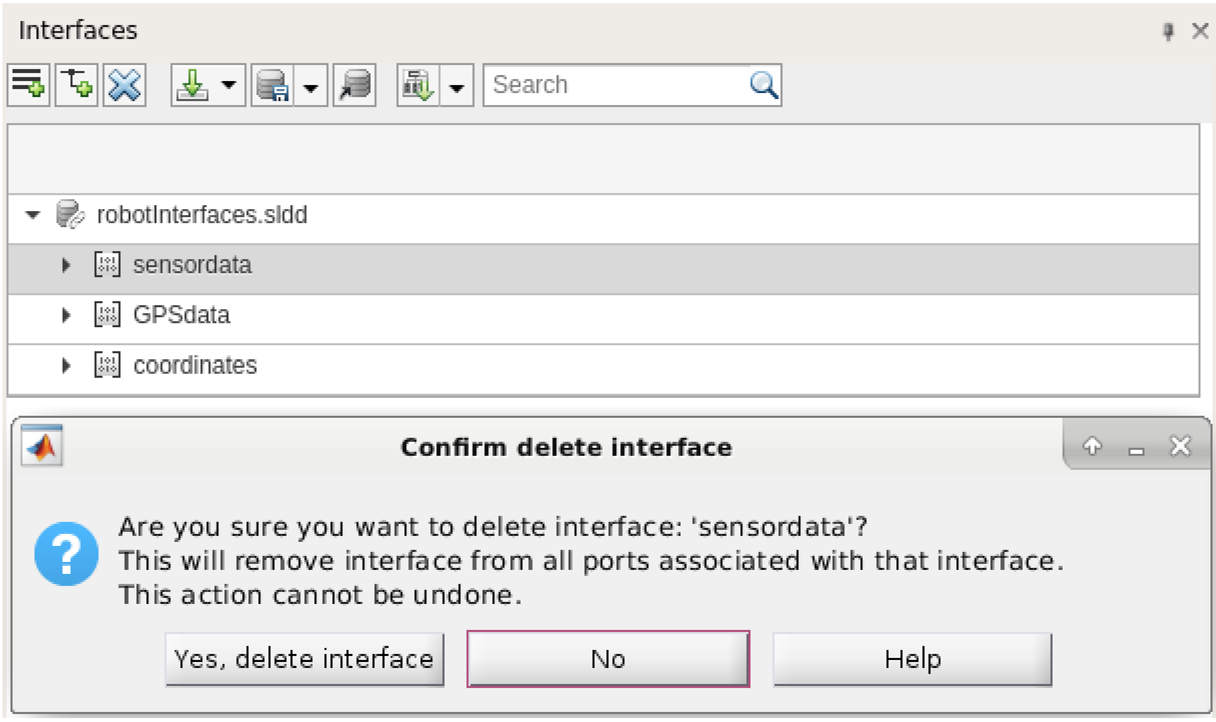
Use the menu to save an interface to a new or existing data dictionary. Create a new data dictionary by selecting **Save to new dictionary**. Provide a dictionary name.



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

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

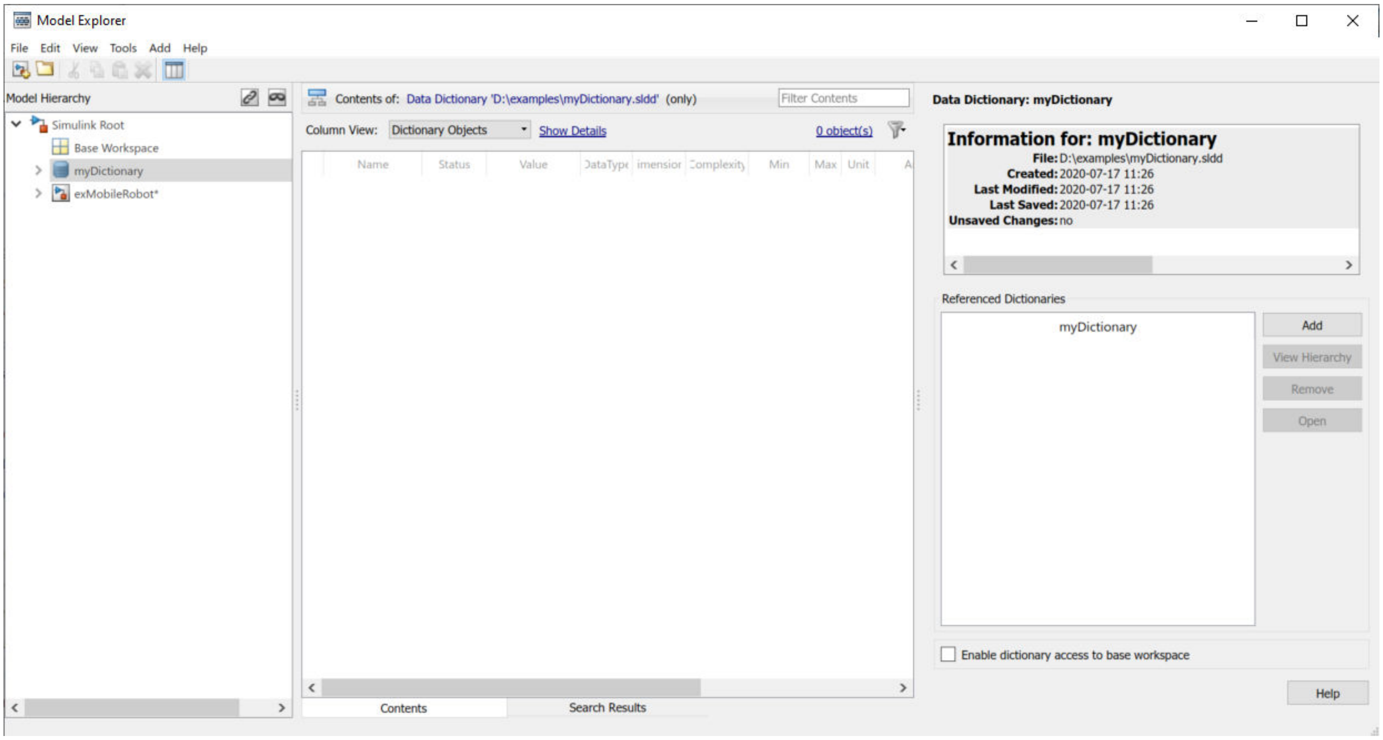
Delete an interface from a dictionary using the button. If the interface is already being used by ports in a currently open model, the software returns a warning message. The interface is then removed from any ports in the open model that are associated with the interface. Note that if an interface is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



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

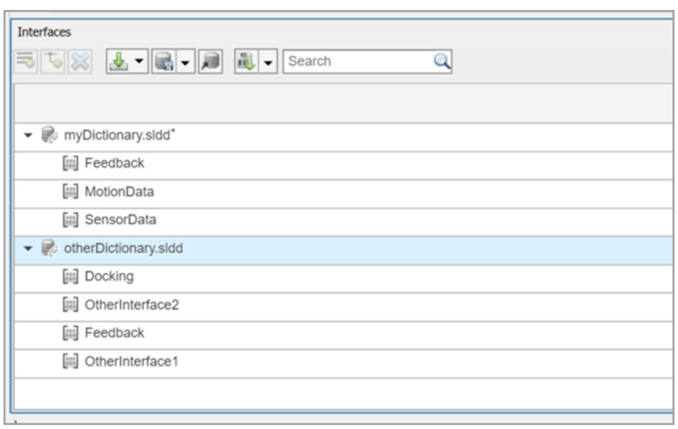
Add Referenced Data Dictionaries

A data dictionary can reference one or more other data dictionaries. The interface definitions in the referenced dictionaries are visible in the parent dictionary and can be used by a model that is linked to the parent dictionary. To add a dictionary reference, open the Model Explorer by clicking on the button, or by selecting **Model Explorer** from the tab in the **Design** section of the **Modeling** tab.



In 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 the following example, `myDictionary.sldd` is the data dictionary linked to the model, and `otherDictionary.sldd` is a referenced dictionary.



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

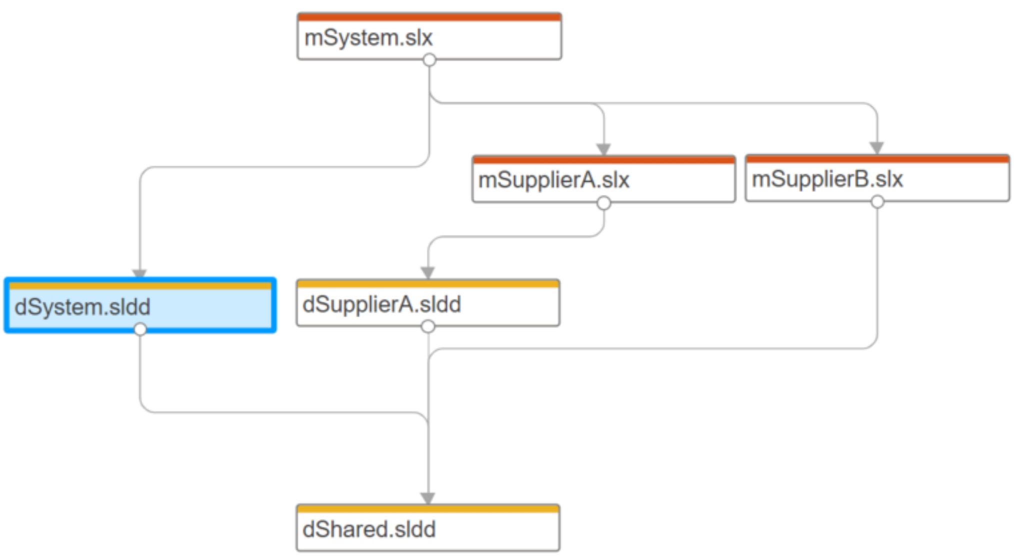
Note that referenced dictionaries can reference other data dictionaries. A model that links to a dictionary has access to all interface definitions in referenced dictionaries, including indirectly referenced dictionaries.

Referenced dictionaries may be useful when multiple models need to share some, but not all, interface definitions. For instance, Model A could link to Dictionary A, which contains interface definitions used only by Model A, and Model B could similarly link to Dictionary B. Both Dictionary A and Dictionary B could then reference Dictionary C, which contains interface definitions shared by both models, for example, to allow communication between the models.

Use Referenced Data Dictionaries for Projects with Multiple Models

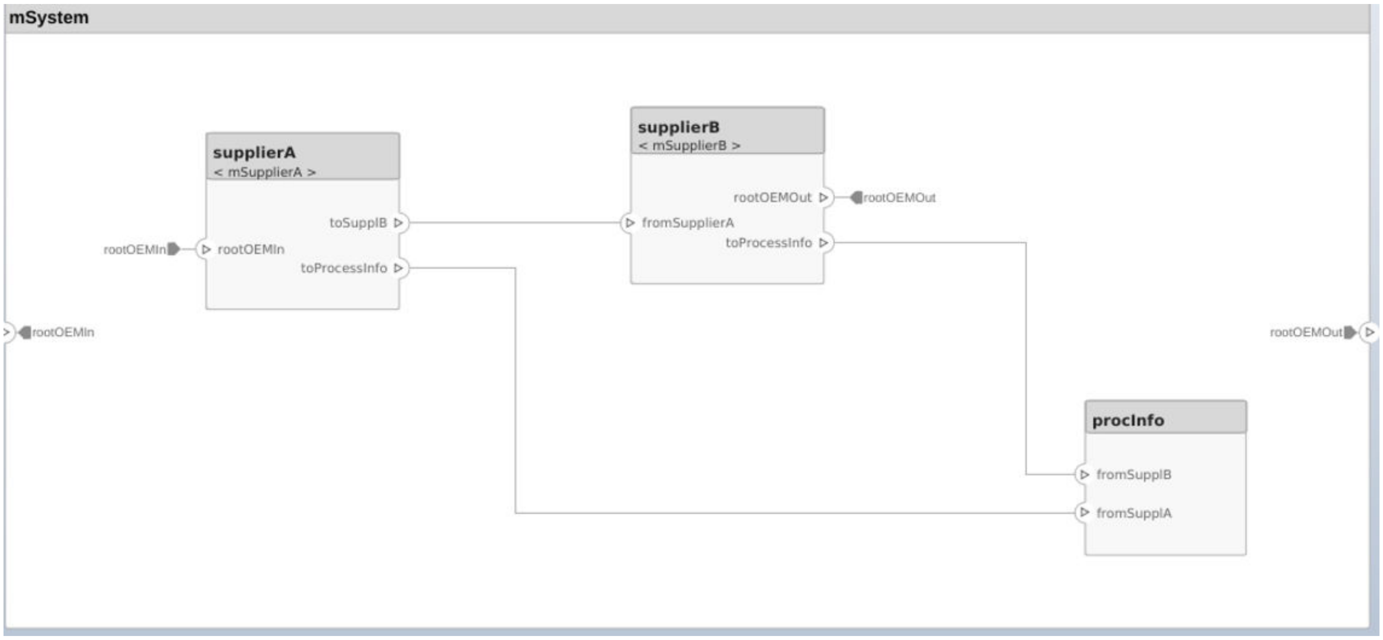
A project may contain multiple models, and it may be useful for the models to share interface definitions that are relevant to data flows and other communications between models. At the same time, each model may have interface definitions that are relevant only to its internal operations. For example,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

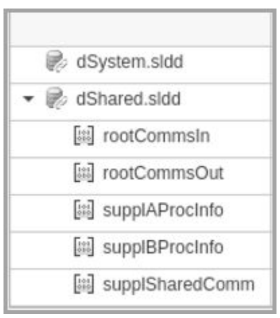
In such a multiple-team project, set up a "shared artifacts" data dictionary to store interface definitions that will be shared by different teams, then set up a data dictionary for each model within the project to store its own interface definitions. Each data dictionary can then add the shared data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



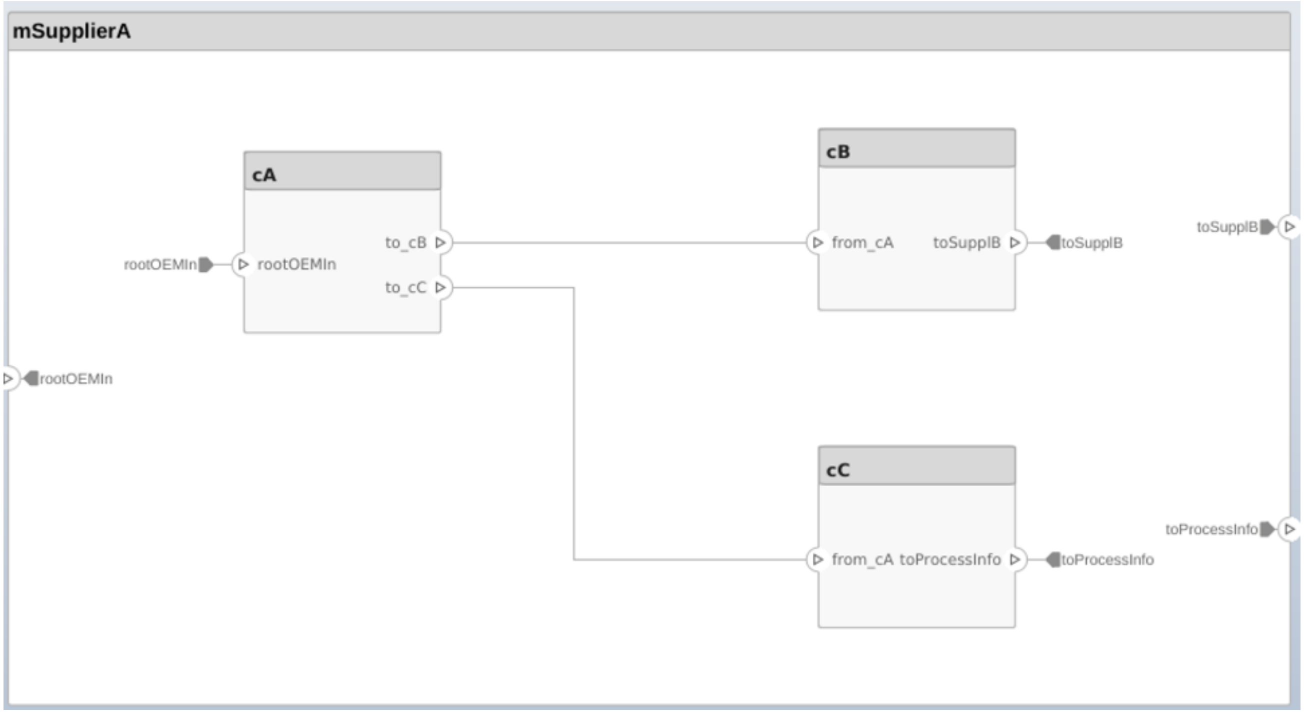
The above diagram depicts a project with three models. The model `mSystem.slx` represents a system integration model, and `mSupplierA.slx` and `mSuppierB.slx` represent supplier models. The data dictionary `dShared.sldd` contains interface definitions shared by all the models. The system integration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

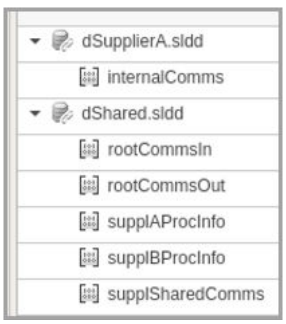
The following diagrams show the system integration model `mSystem`, along with the Interface Editor. Interface definitions contained in the referenced dictionary `dShared` are associated with the ports used to communicate between the models `mSupplierA` and `mSupplierB` and the rest of the system integration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





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.





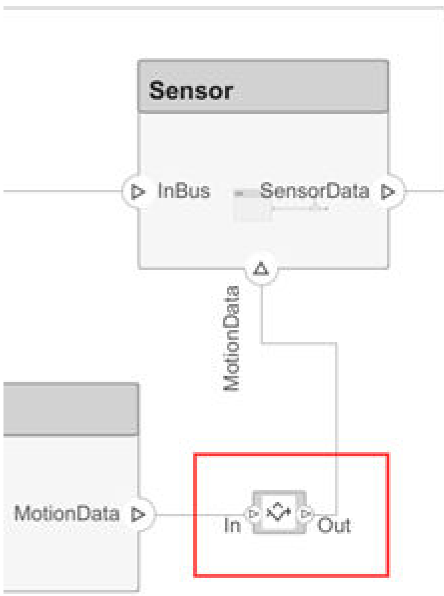
See Also

More About

- “Define Interfaces” on page 3-2
- “Assign Interfaces to Ports” on page 3-5

Interface Adapter

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



Map Similar Interfaces

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

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

Use Unit Delay to Break Algebraic Loop

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

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

Use Rate Transition Between Simulink Behaviors

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

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

See Also

Blocks
Adapter

More About

- “Define Interfaces” on page 3-2
- “Save Simulink.Bus Objects”
- “Assign Interfaces to Ports” on page 3-5

Define Architectural Properties

Define Profiles and Stereotypes

To verify structural and functional requirements, you must capture nonfunctional properties on elements in an architecture model. For example, if there is a limit on the total power consumption of a system, the model must capture the power rating of each electrical component. This requires extending built-in model element types with properties corresponding to requirements, in this case, an electrical component type as an extension of components. You can introduce a self-consistent domain of model element types into System Composer using a group of property sets, or stereotypes, called a profile.

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

- Component
- Port
- Connection

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

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

When you create a profile, you define:

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

You can define stereotypes to extend built-in elements and capture additional data about an element. Element stereotypes define the class of the elements to which they apply. For example, a `MechanicalComponent` stereotype with properties such as `Weight` and `Volume` applies only to components.

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

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

You can collect stereotypes in profiles.

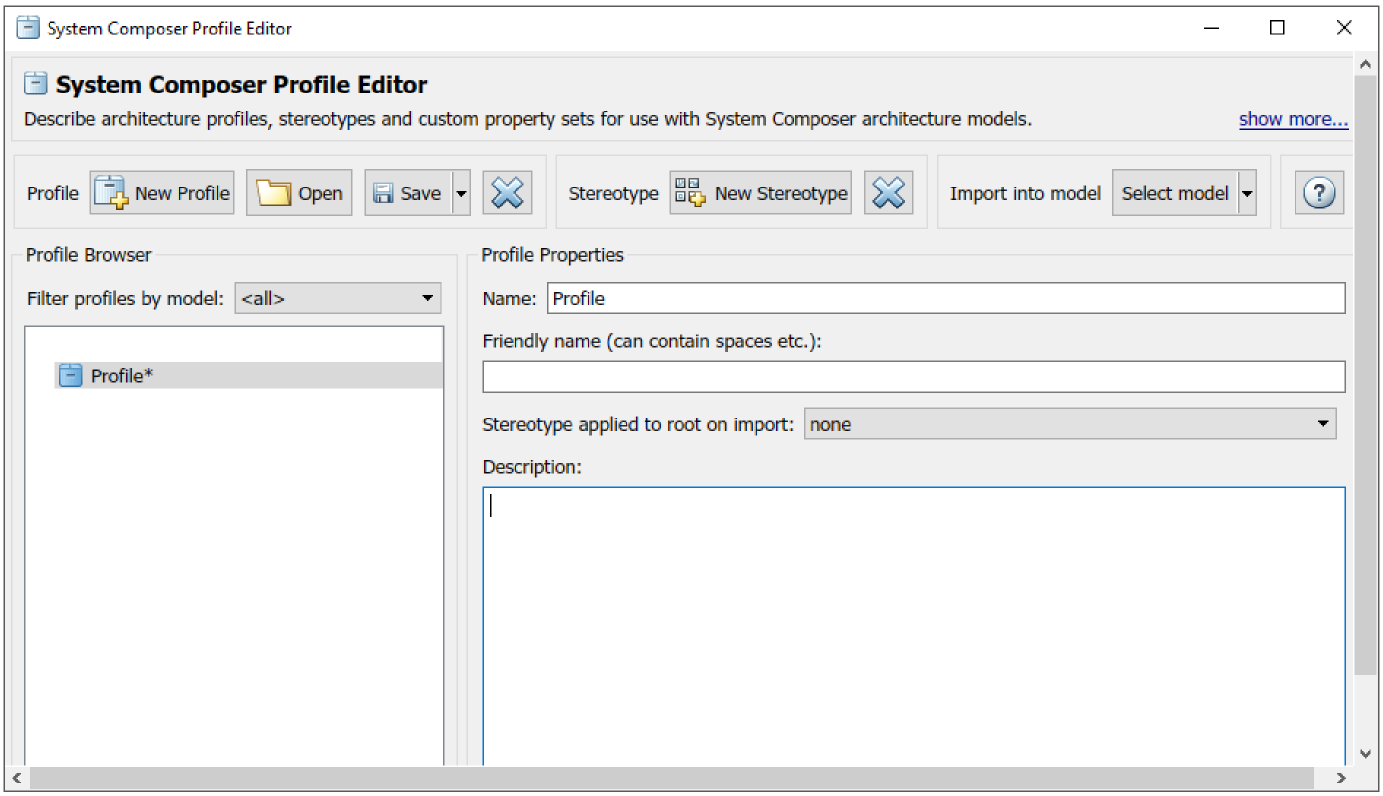
Create a Profile and Add Stereotypes

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. In any architecture model, select **Architecture > Profile > Profile Editor**. Click **New Profile**. Select new profile to start editing.



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

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

Add Properties with Stereotypes

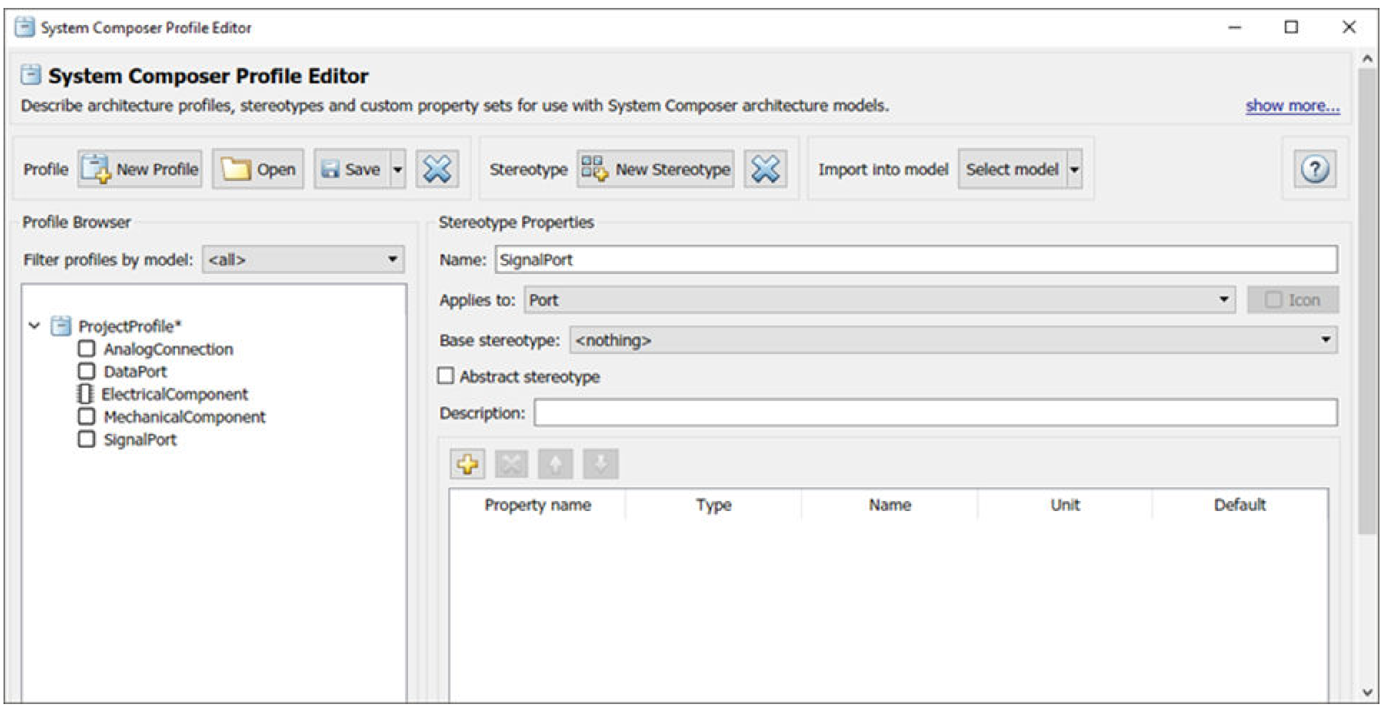
Select a stereotype in a profile to define it:

- **Name** — The name of the component type, for example, `ElectricalComponent`.

- **Applies to** — The model element type to which the stereotype applies. This field can be an architecture, component, port, connector, or interface. You can apply this stereotype only to a model element of this type.
- **Icon** — Icon to be shown on the model element.
- **Base stereotype** — Other stereotype on which this stereotype is based. This can be empty.
- **Abstract stereotype** — A stereotype that is not intended to be applied directly to a model element. You can use abstract stereotypes only as the base stereotype for other stereotypes.

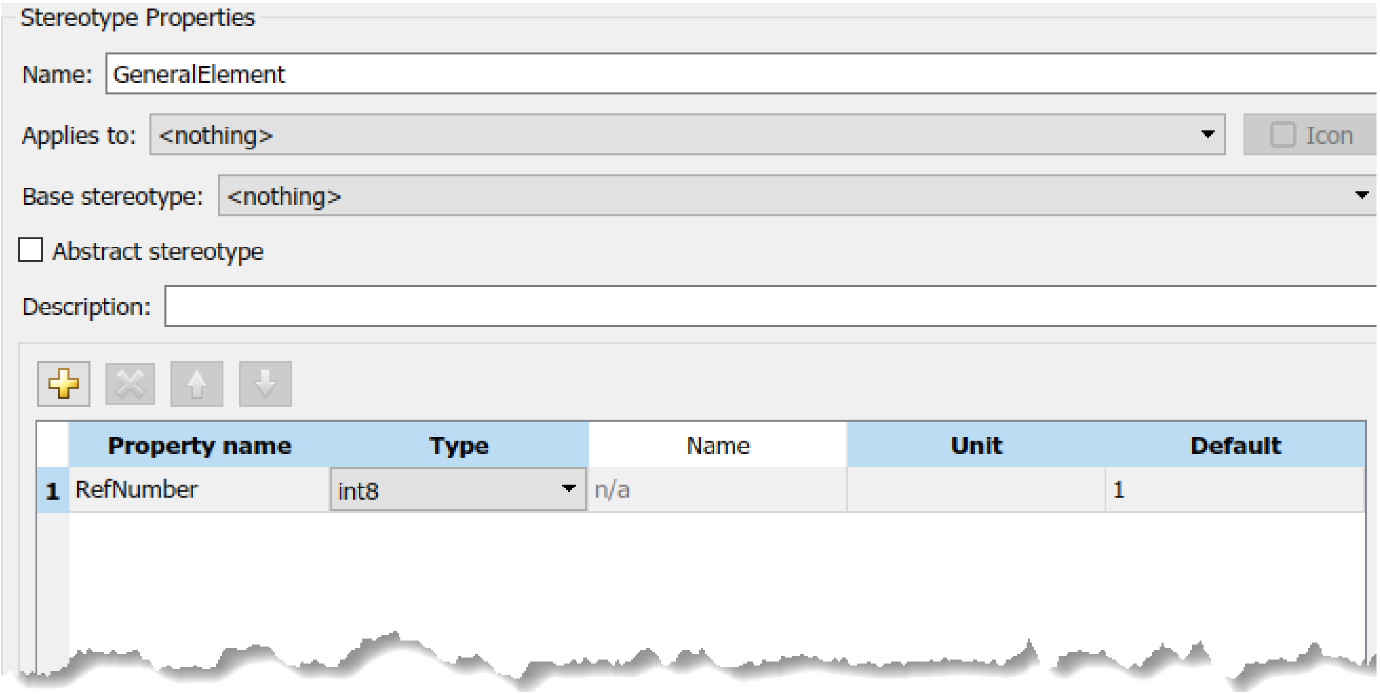
Add properties to a stereotype using . Define these fields for each property:

- Property name — Valid variable name
- Type — Numerical, string, or enumeration data type
- Unit — Value units as a string
- Default — Default value



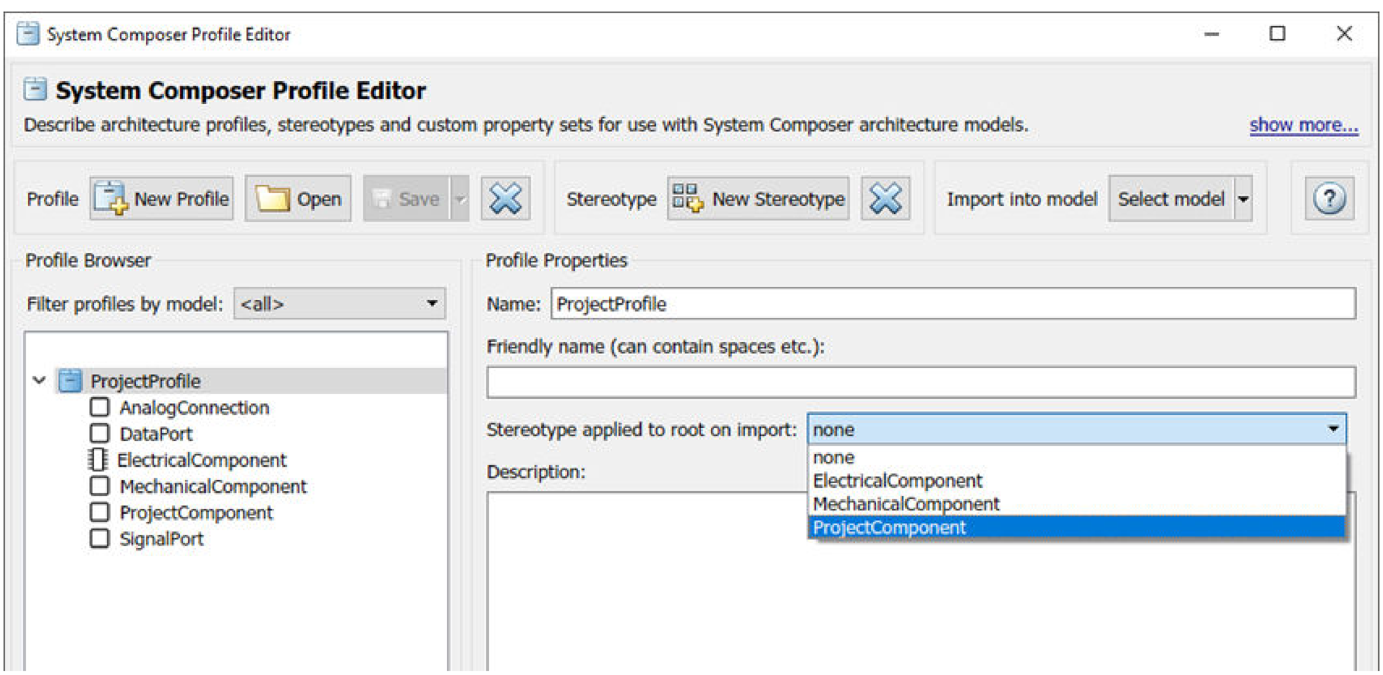
Add, delete, and reorder properties using the property toolstrip: 

You can create a stereotype that applies all model element types by setting the **Applies to** field to **<nothing>**. With these stereotypes, you can add properties to elements regardless of whether they are components, ports, connectors, or architectures.



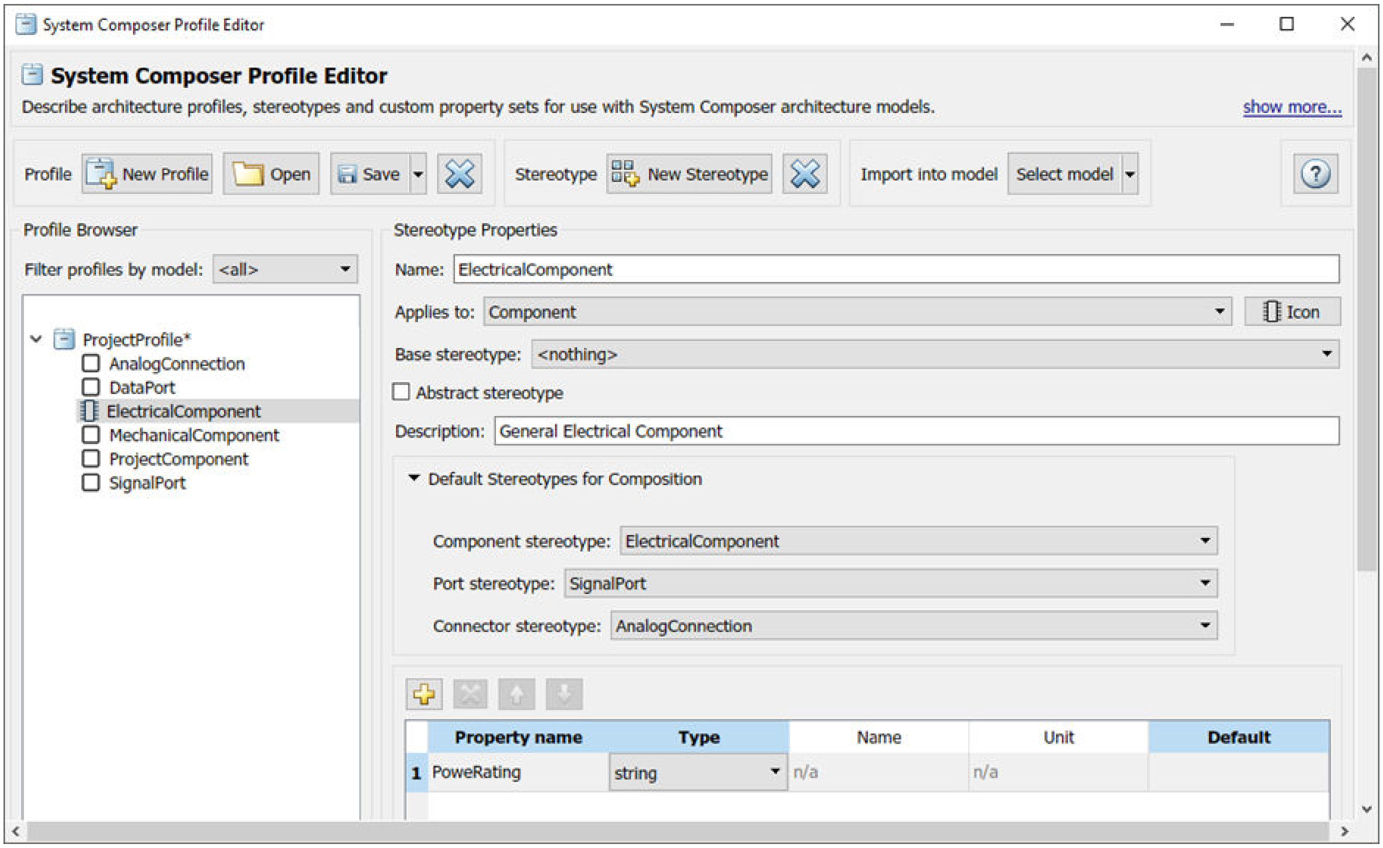
Default Stereotypes

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



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

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

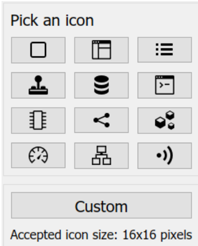


After you import the profile into a model, all new connections assume the `AnalogConnection` stereotype.

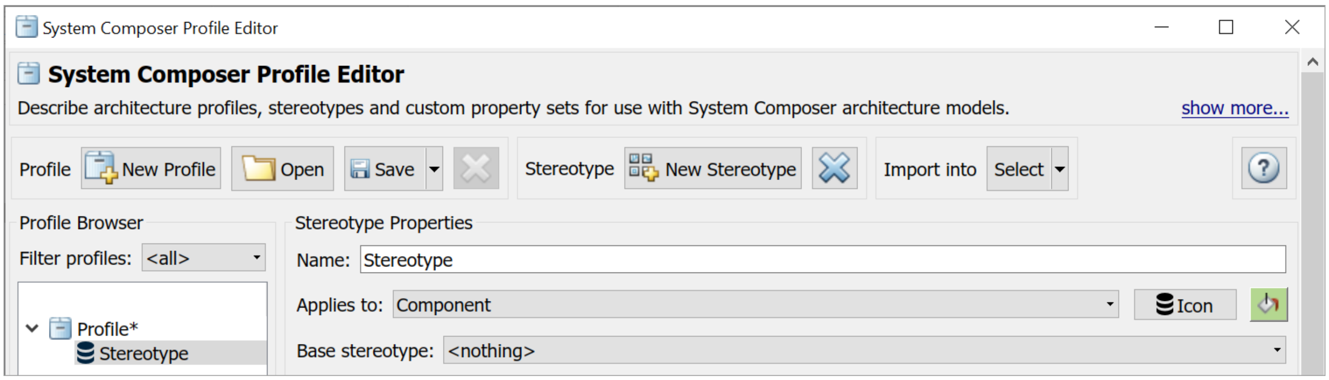
Stereotype-Based Styling

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

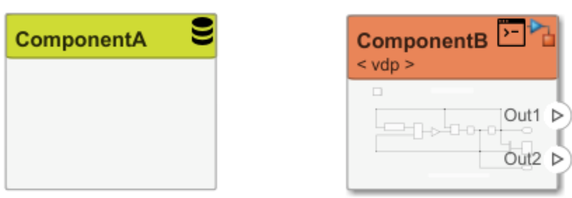
You can use provided icons for the component stereotypes or use you own custom icon images. Custom icons support `.png`, `.jpeg`, or `.svn` image files of size 16-by-16 pixels. The custom icons are displayed as badges on the components for which the stereotypes are applied.



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. You can use a preconfigured set of color options for component stereotypes to style the architecture component headers. Below is an example that displays the applied component stereotypes with icons and color. See “Use Stereotypes and Profiles” on page 4-9 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.

See Also

"Use Stereotypes and Profiles" on page 4-9

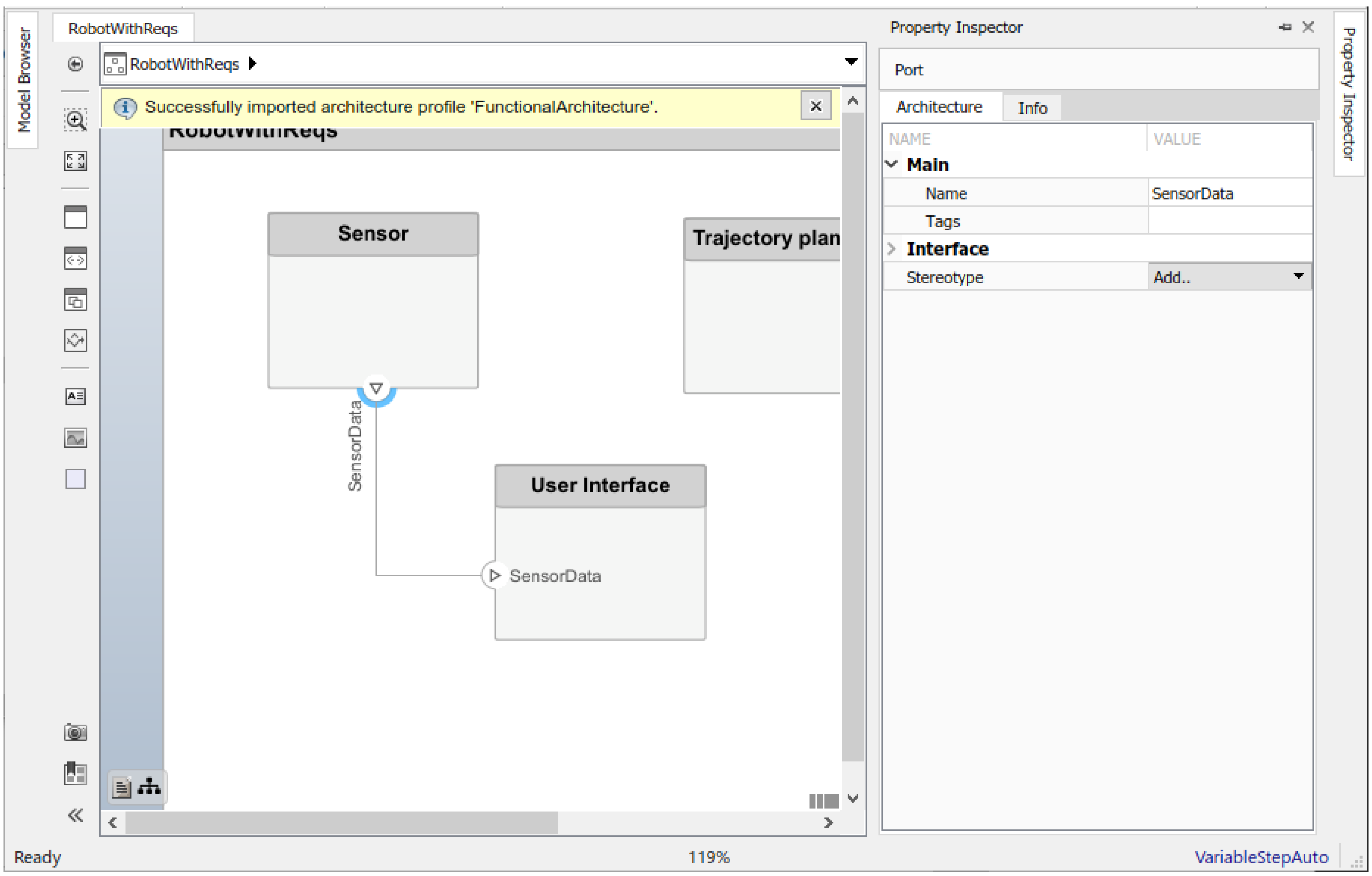
Use Stereotypes and Profiles

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

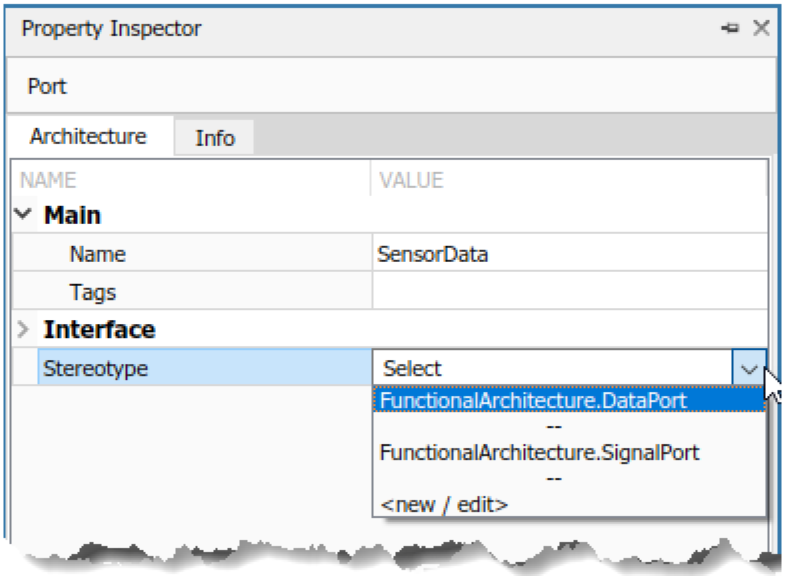
Apply a Stereotype

The Profile Editor is independent from the model that opens it, that is, you must explicitly import a new profile into a model. On the **Model** tab and in the **Profiles** section, select **Manage** and then from the drop-down, select **Import** . Select the profile to import. An architecture model can use multiple profiles at once.

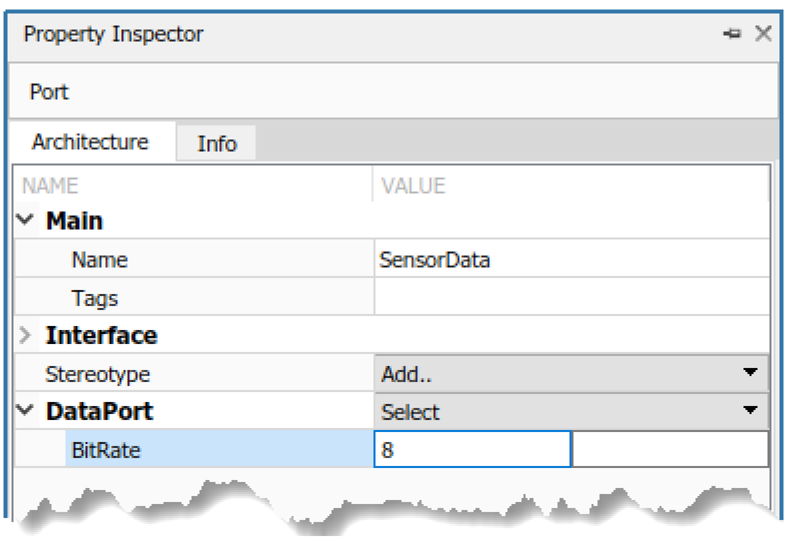
Once the profile is available in the model, open the Property Inspector. On the **Modeling** tab and in the **Design** section, select **Property Inspector**. Select a model element.



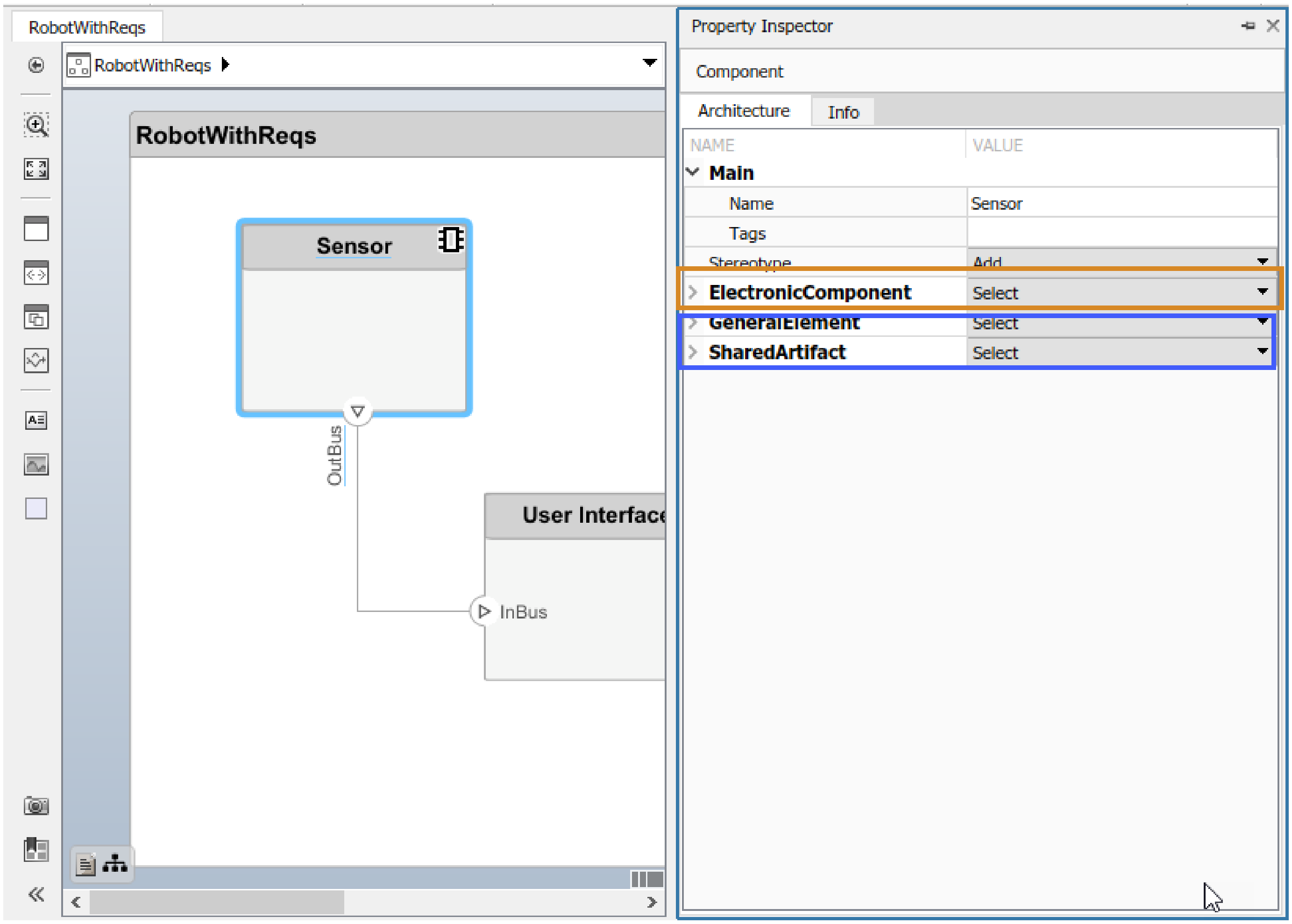
In the **Stereotype** field, use the drop-down to select the stereotype. Only the stereotypes that apply to this element type (for this example, a port) are available for selection. If no stereotype exists, you can use the **<new/edit>** option to open the profile editor and create one.



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



You can set multiple stereotypes for each element.

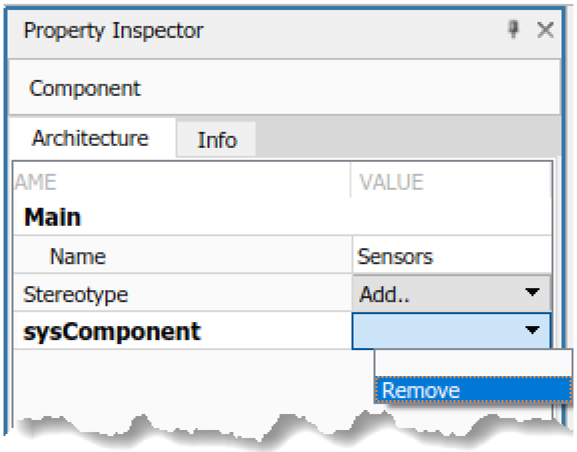


You can also apply component and connector stereotypes to all applicable elements at the same level.

On the **Modeling** tab and in the **Profiles** section, select **Apply Stereotypes**. In the Apply Stereotypes dialog box and from the **Apply to** list, select `All elements`, `Components`, `Ports`, or `Connectors`. From the **in** list, select `Selection`, `This layer`, or `Entire model`.

Remove a Stereotype

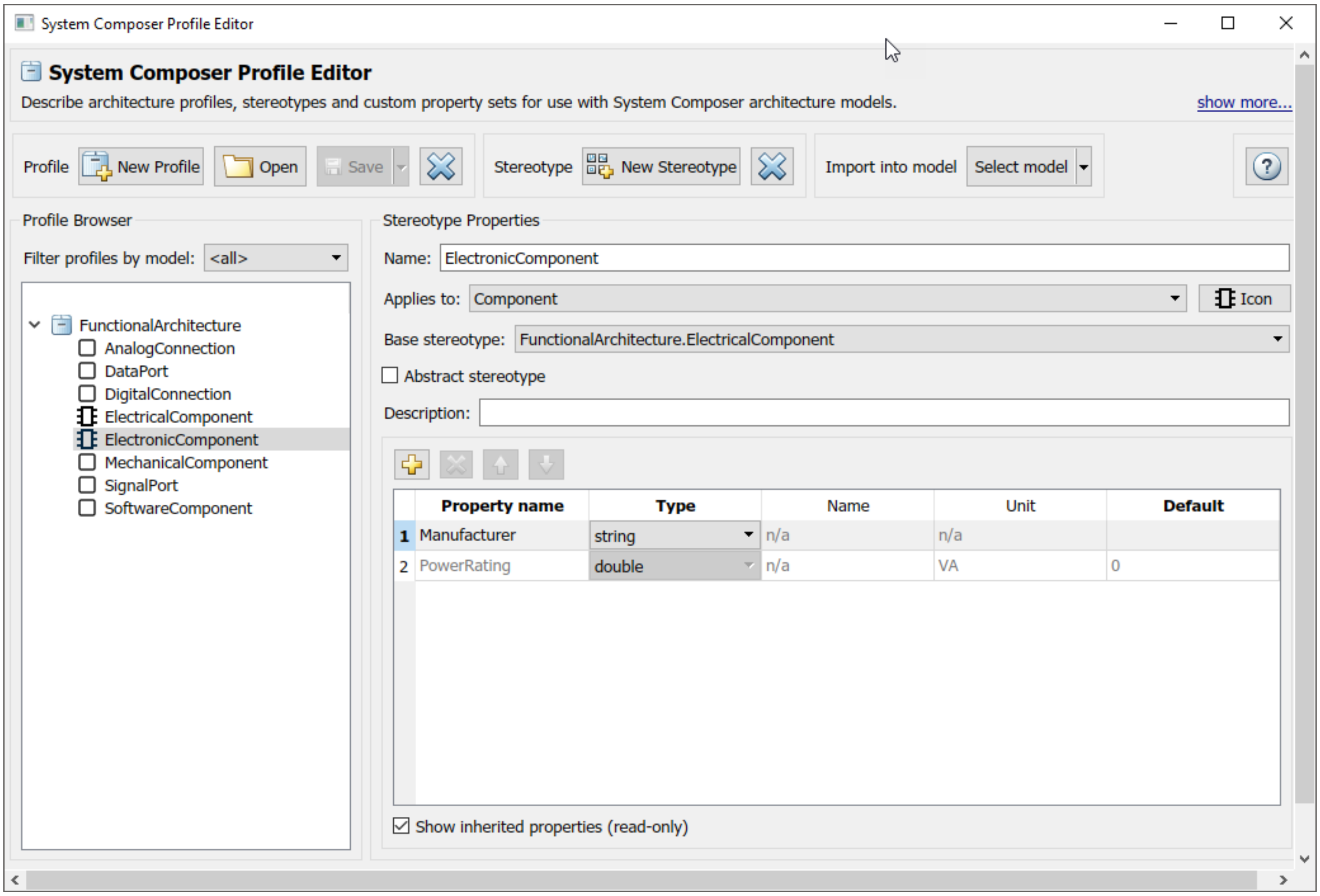
If a stereotype is no longer required for an element, remove it using the Property Inspector. Click **Select** next to the stereotype and choose **Remove**.



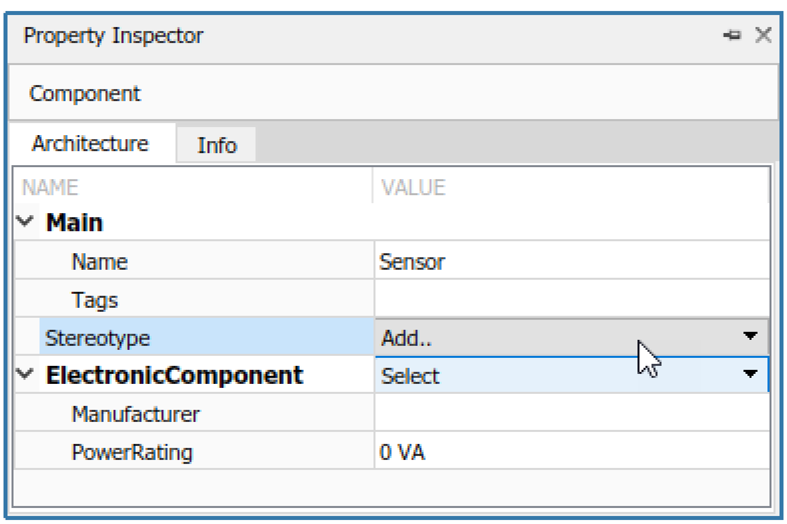
Extend a Stereotype

You can extend a stereotype by creating a new one based on the existing one. This allows 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, which is 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

More About

- "Define Profiles and Stereotypes" on page 4-2
- "Analyze Architecture" on page 6-9

Use Simulink Models with System Composer

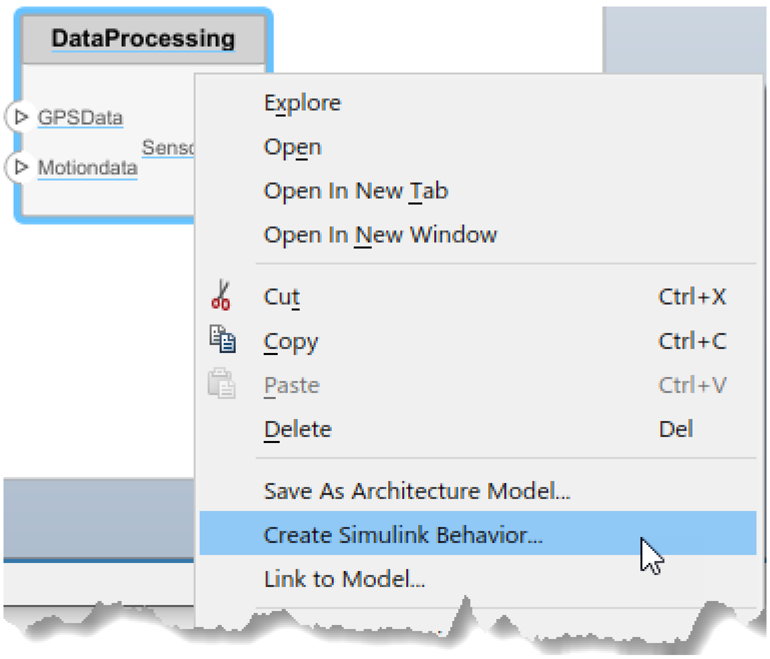
Implement Components in Simulink

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

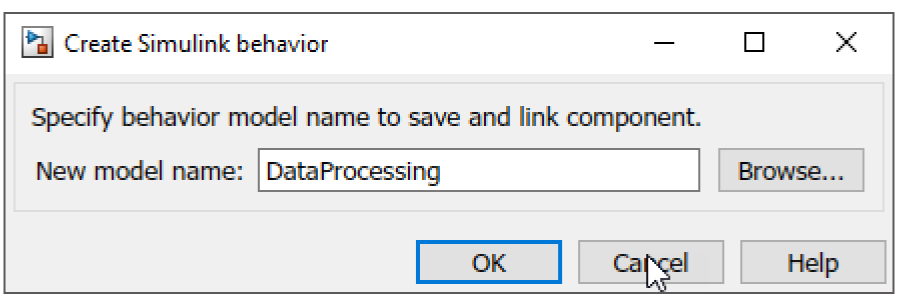
Create a Simulink Behavior Model

When a component does not require further decomposition from an architecture standpoint, you can design and define its behavior in Simulink.

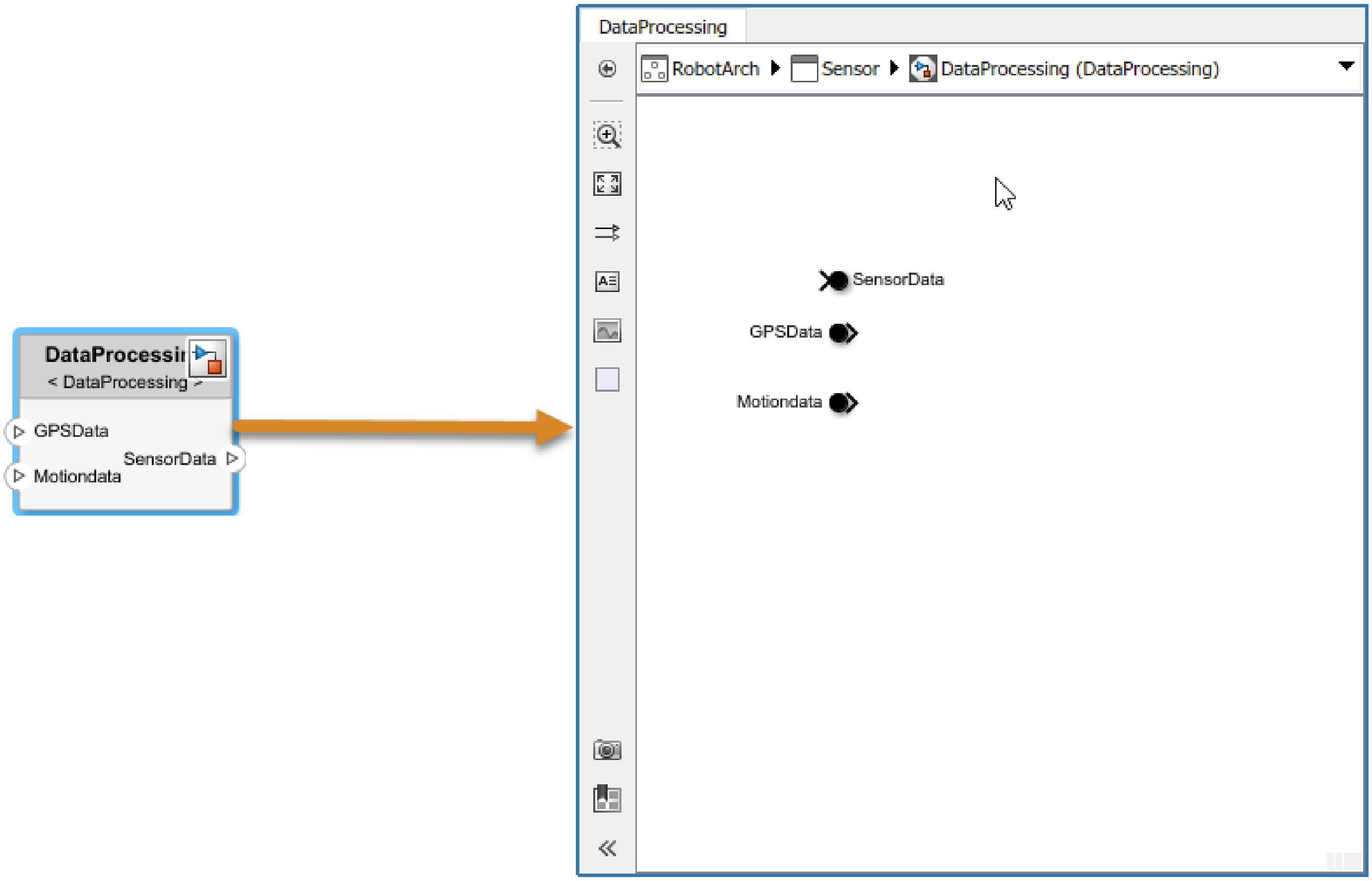
1 Right-click the component and select **Create Simulink Behavior**.



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

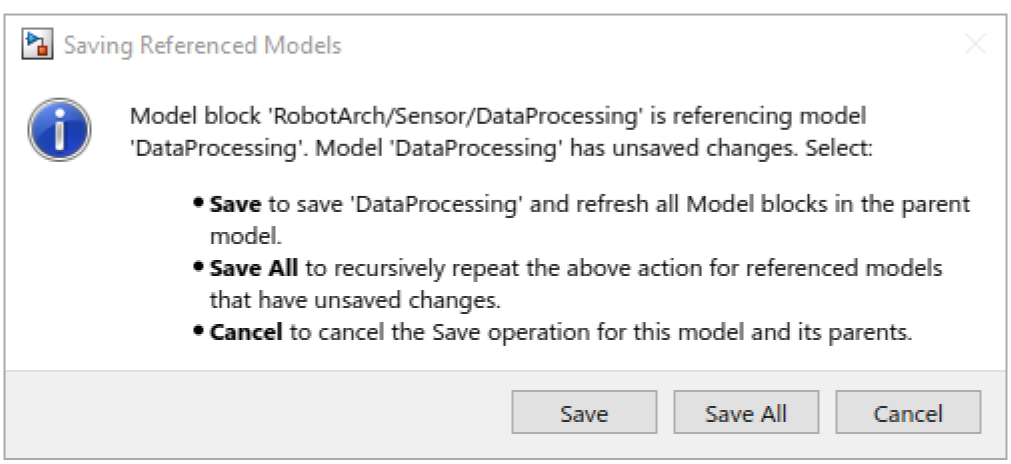


- A new Simulink model with the provided name is created. The root level ports of the Simulink model reflect the ports of the component.
- The component in the architecture model is linked to the Simulink model. The Simulink icon on the component indicates this is a Simulink link.



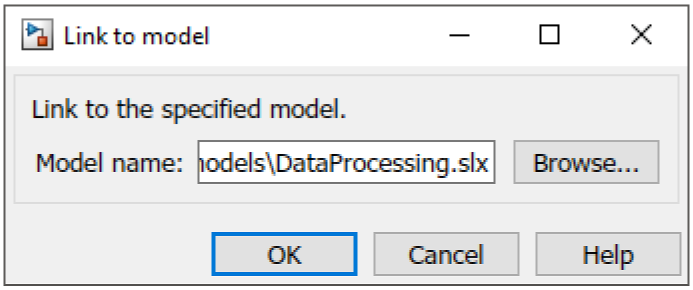
You can continue with providing specific dynamics and algorithms in the referenced Simulink model. Adding root-level ports in the Simulink model creates additional ports on the System Composer Reference Component block.

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



Link to an Existing Simulink Behavior Model

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



Any subcomponents and ports that are present in the components get deleted when the component links to a Simulink model.

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

See Also

More About

- “Decompose and Reuse Components” on page 1-15
- “Extract Architecture from Simulink Model” on page 5-5

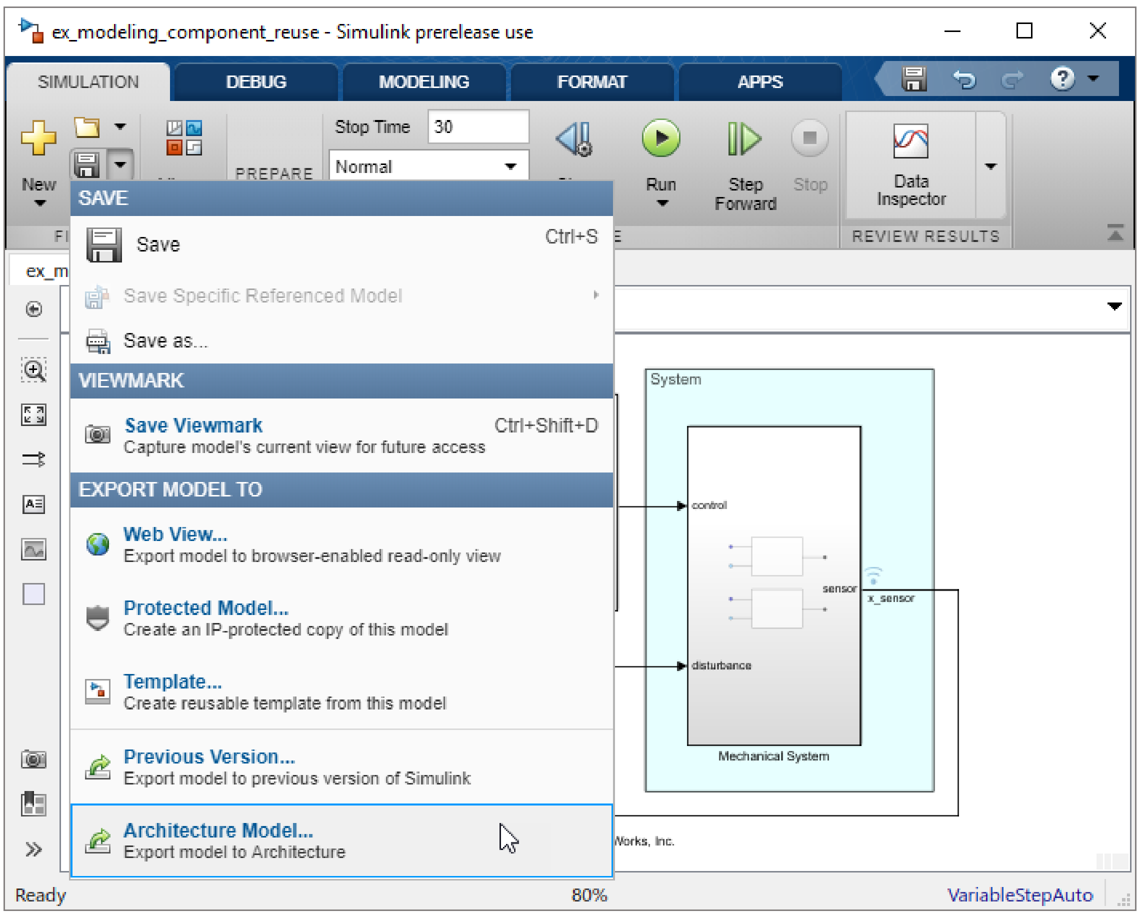
Extract Architecture from Simulink Model

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.

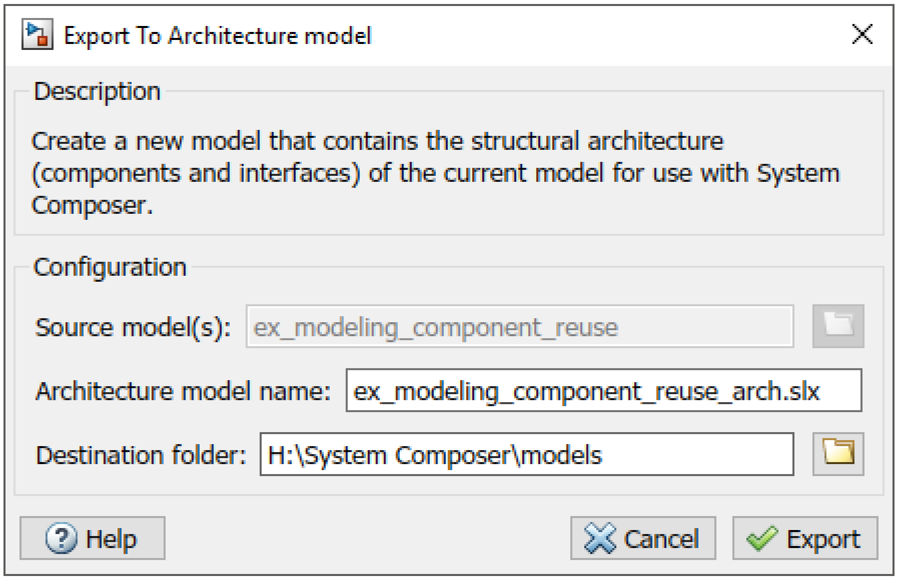
1 Open an example model.

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

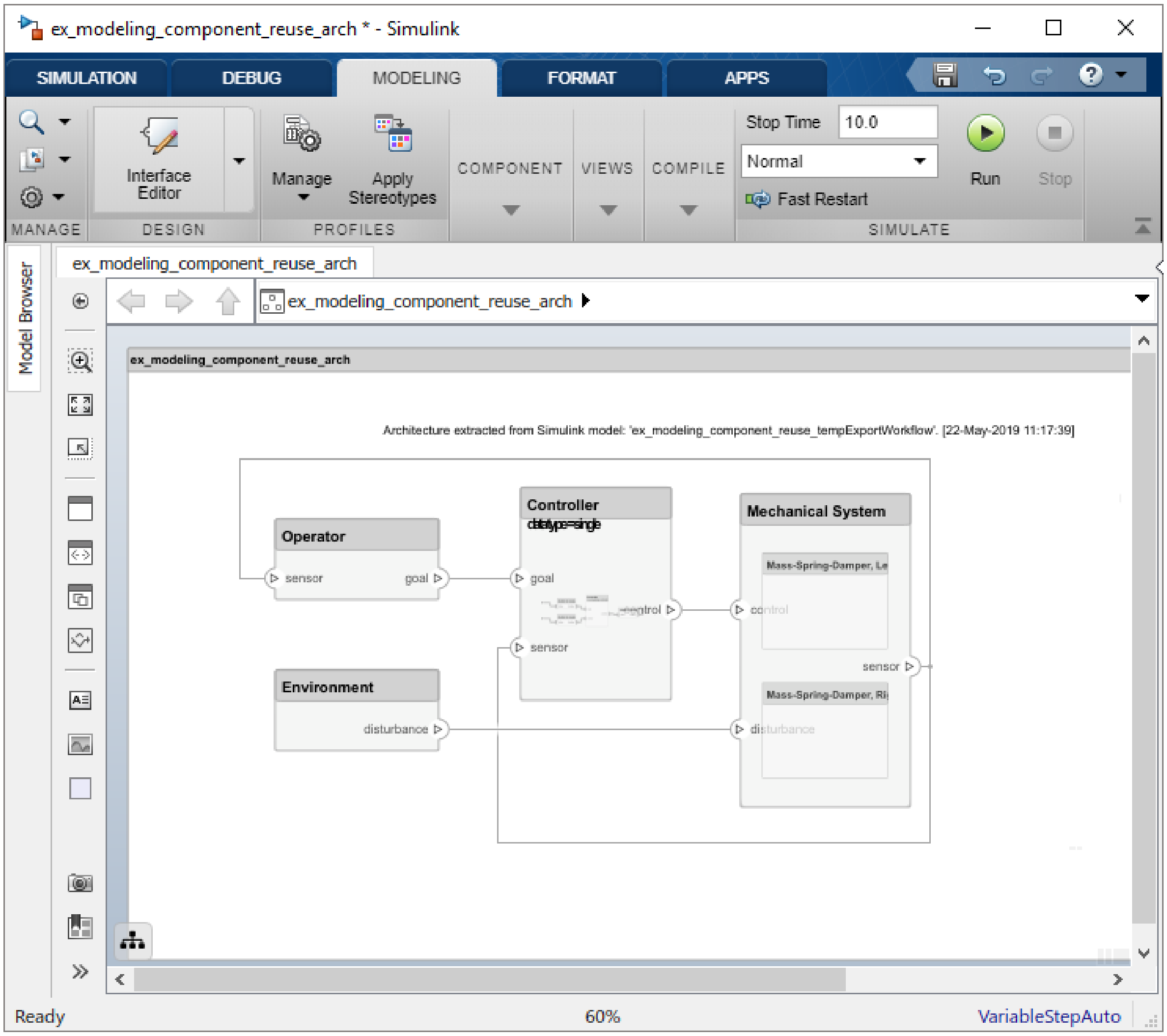
2 On the **Simulation** tab, click the **Save** arrow. From the **Export Model To** list, select **Architecture Model**.



3 Provide a name and path for the architecture model.



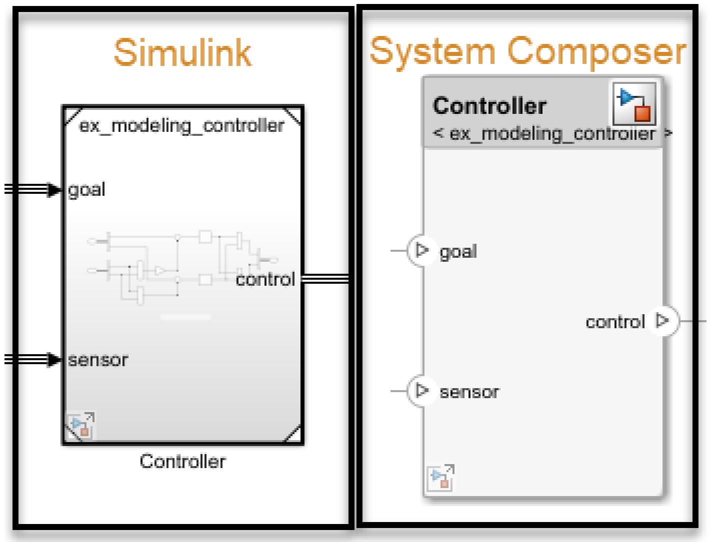
4 Click **Export**. A System Composer Editor window opens with an architecture model corresponding to the Simulink Model.



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

More About

- “Implement Components in Simulink” on page 5-2
- “Decompose and Reuse Components” on page 1-15

Analyze Architecture Model

Create and Manage Allocations

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

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

```
scExampleTirePressureMonitorSystem
```

Create a New Allocation Set

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

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

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

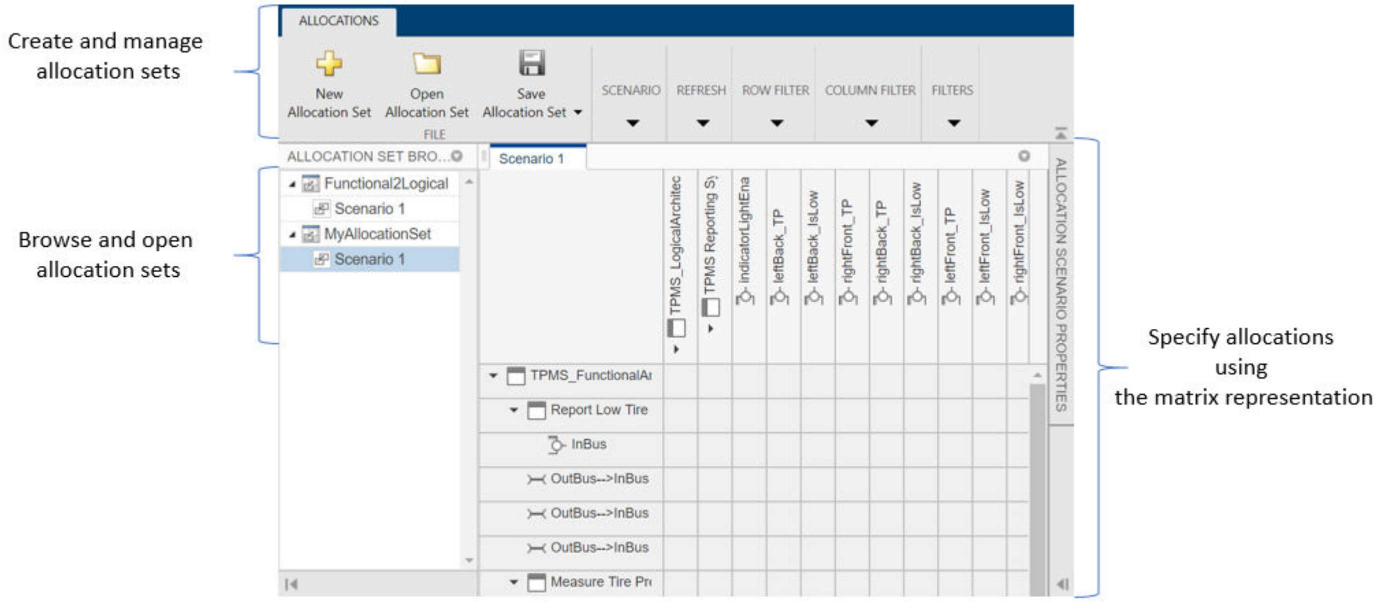
```
allocSet = systemcomposer.allocation.createAllocationSet(...
    'Functional2Logical', ...% Name of the allocation set
    'TPMS_FunctionalArchitecture', ... % Source model
    'TPMS_LogicalArchitecture' ... % Target model
     );
```

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

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

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

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



Create Allocations between Two Models

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

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

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

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

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

Create the allocations in the default scenario that is created.

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

Save the allocation set.

```
allocSet.save;
```

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

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

See Also

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

More About

- “Allocate Architectures in a Tire Pressure Monitoring System” on page 6-5

Allocate Architectures in a Tire Pressure Monitoring System

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

Overview

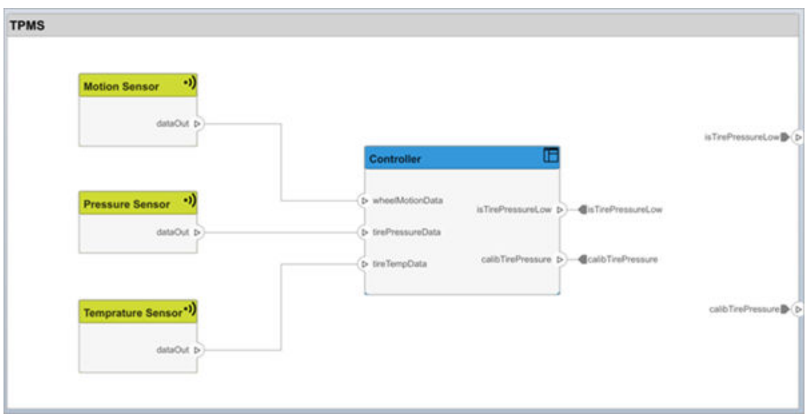
In Systems Engineering, it is common to describe a system at different levels of abstraction. For example, you can describe a system in terms of its high-level functions. These functions may not have any behavior associated with them but most likely trace back to some operating requirements the system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

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

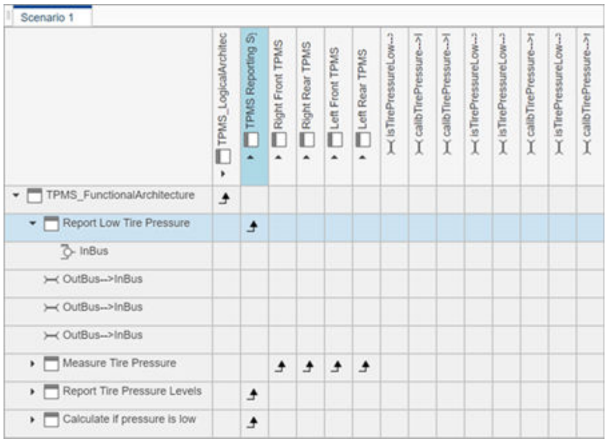
The allocation process is defined as linking these three architectures that fully describe the system. The linking captures the information each architectural layer and makes it accessible to the others.

Use this command to open the project.

```
scExampleTirePressureMonitorSystem
```



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



This figure displays allocations in the architectural component level. The arrows display allocated components in the model. You can observe allocations for each element in the model hierarchy.

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

Functional to Logical Allocation and Coverage Analysis

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

To start the analysis, load the allocation set.

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

Verify that each function in the system is allocated.

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

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

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

Analyze Suppliers Providing Functions

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

```
suppliers = {'Supplier A', 'Supplier B', 'Supplier C', 'Supplier D'};
functionNames = arrayfun(@(x) x.Name, allFunctions, 'UniformOutput', false);
numFunNames = length(allFunctions);
numSuppliers = length(suppliers);
allocTable = table('Size', [numFunNames, numSuppliers], 'VariableTypes', repmat("double", 1, numS
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 =

8×4 table

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

function memoryUsed = getUtilizedMemoryOnECU(ecu, scenario)
% For each of the components in the ECU, accumate the binary size
% required for each of the allocated software components.

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

end
```

```
softwareDeploymentTable =

  6×2 table
```

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

See Also

getAllocatedFrom | getAllocatedTo | getScenario | load

More About

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

Analyze Architecture

Write 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:

- Define a profile containing a set of property sets that describe some analyzable properties (for example, cost and weight)
- Apply the profile to an architecture model and add property sets from that profile to elements of the model (components, ports, or connectors)
- Specify values for the properties on those elements
- 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
- Write an analysis function to compute values necessary for the study
- Run the analysis function

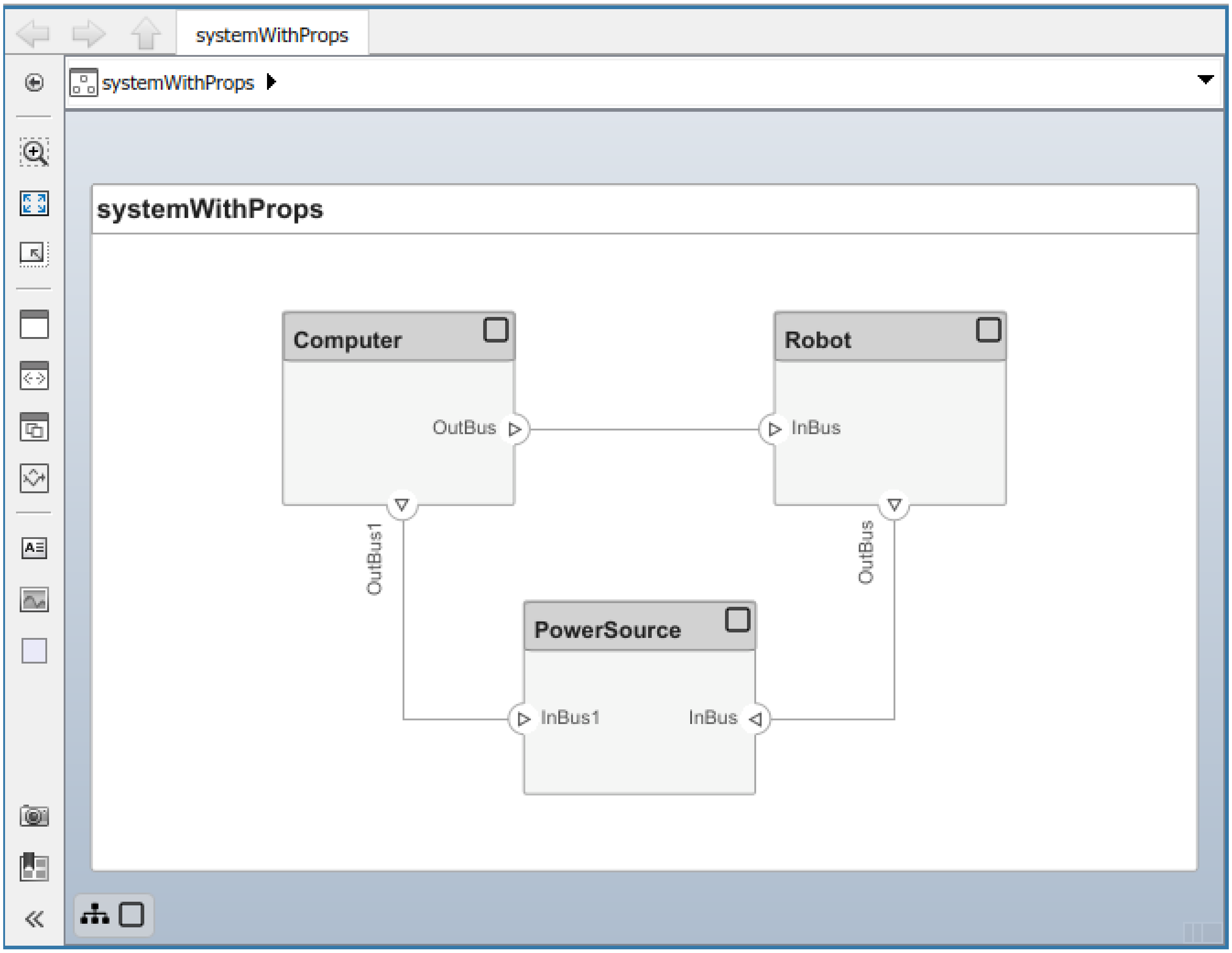
Set Tags and Properties for Analysis

Enable analysis by tagging model elements and setting property values.

Load the Model

Open the `systemWithProps` model.

```
systemWithProps
```



Import a Profile

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

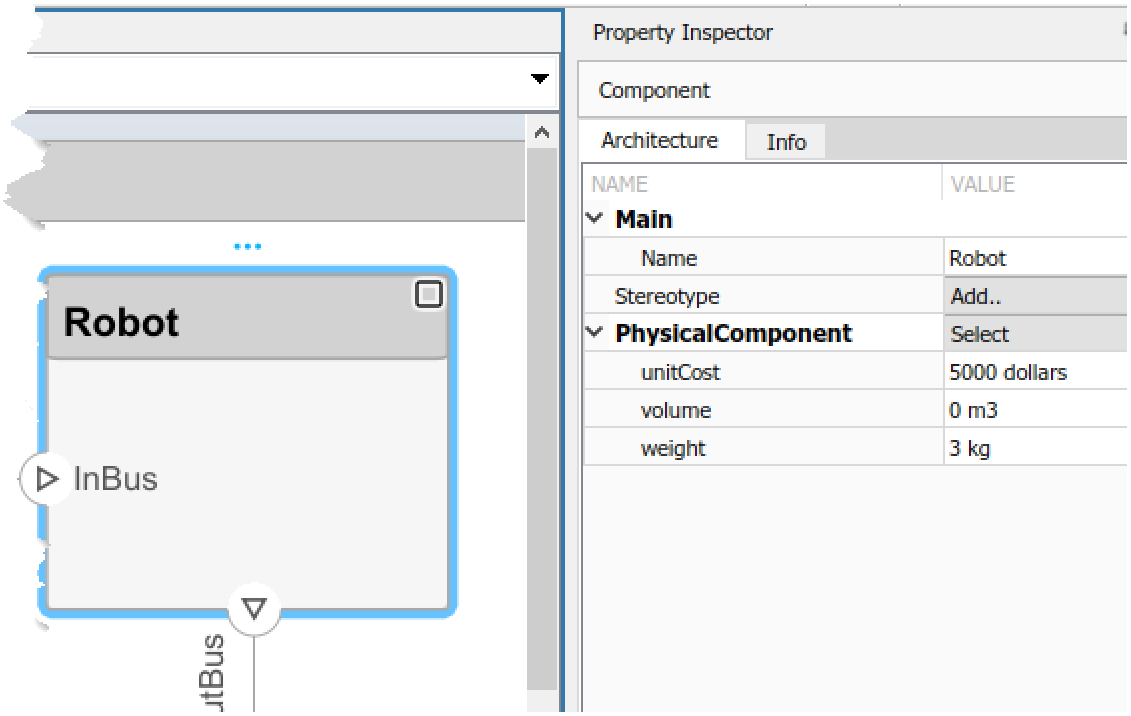
Apply Stereotypes to Model Elements

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

Set Property Values

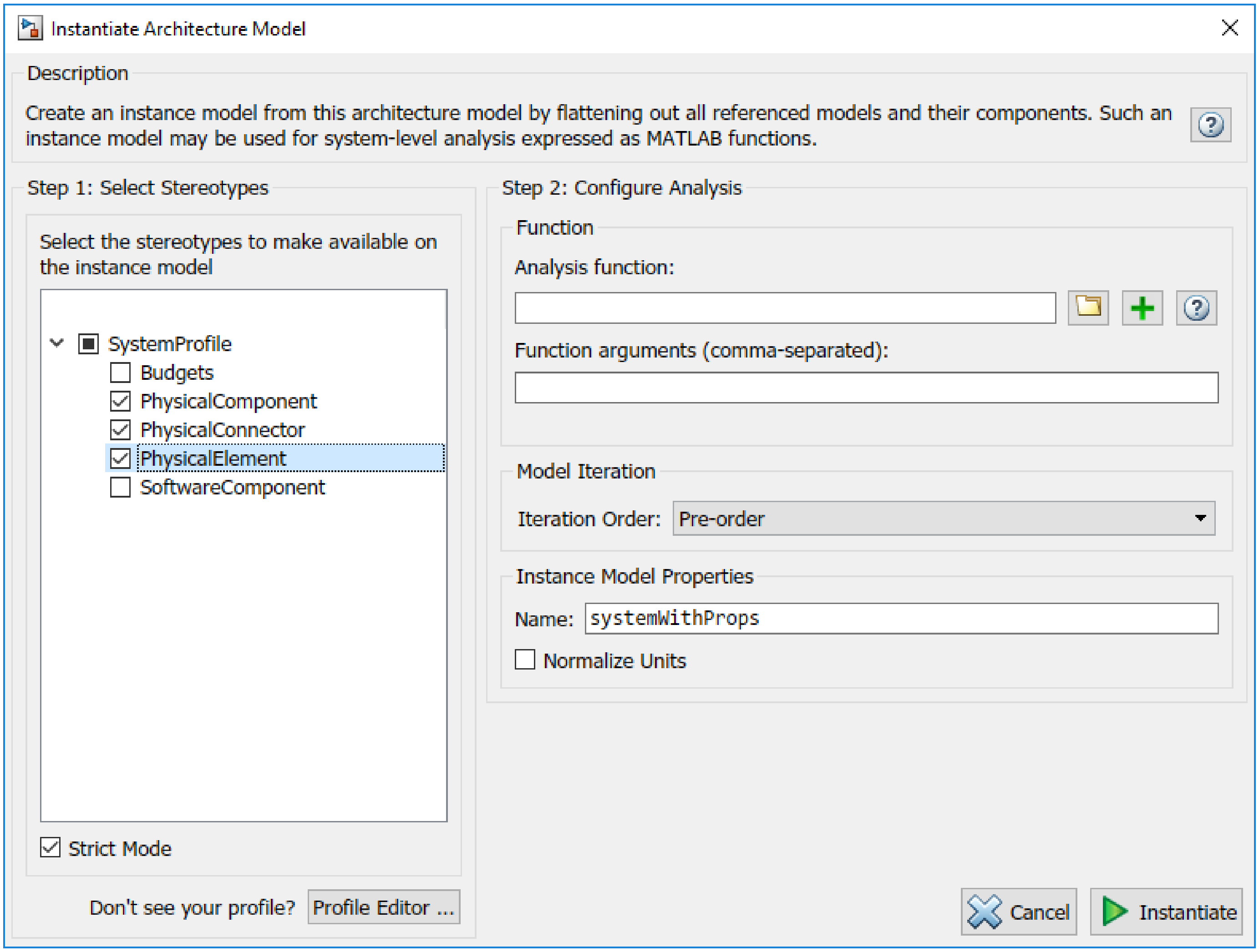
Set property values for each model element.

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



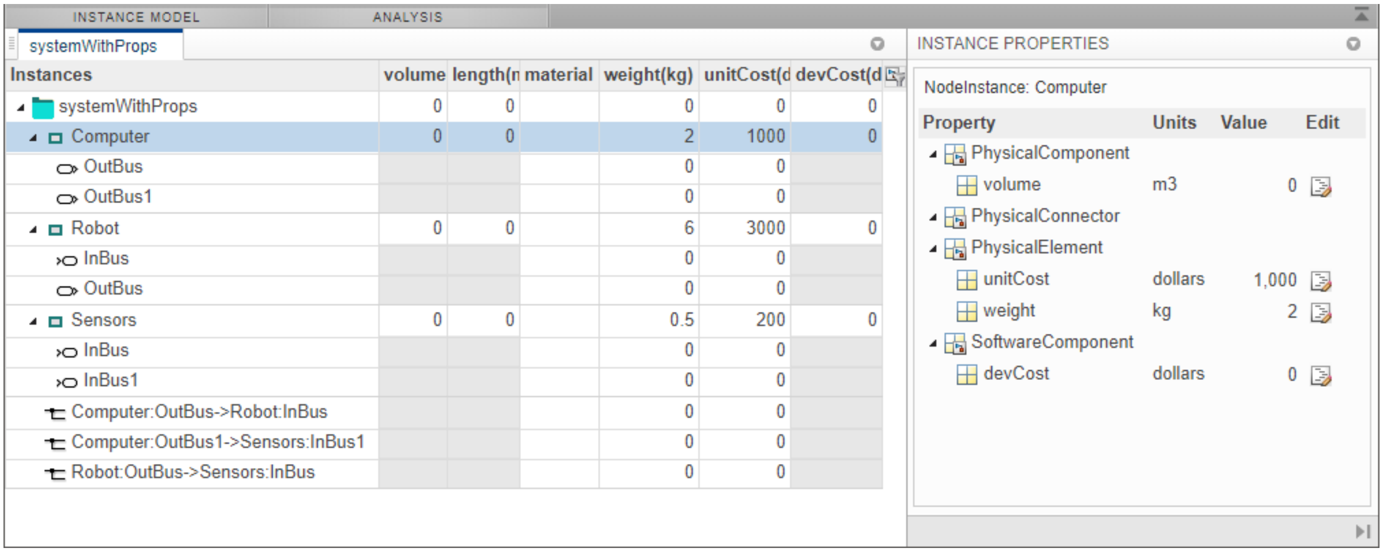
Create a Model Instance for Analysis

Create an instance of the architecture model that you can use for analysis. In the **Views** section, select **Analysis Model** > **Analysis Model**. In this dialog box, specify all the parameters required to create and view an analysis model.



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

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



The Analysis Viewer shows all components, ports, and connectors in the first column. The other columns are properties for all stereotypes chosen for this instance. If a property is not part of a stereotype applied to an element, that field is greyed out. You can use the Filter button to hide properties for certain stereotypes. When you select an element, Instance Properties shows its stereotypes and property values. You can save an instance in a MAT-file, and open it again in the Analysis Viewer. If you make changes in the model while an instance is open, you can synchronize the instance with the model by clicking Update. Unsynchronized changes are shown in a different color.

Write Analysis Function

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

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

```
function systemWithProps_1(instance,varargin)
% systemWithProps_1 Example Analysis Function
if instance.isComponent()
    sysComponent_unitPrice = instance.getValue("PhysicalElement.unitCost");
    for child = instance.Components
        comp_price = child.getValue("PhysicalElement.unitCost");
        sysComponent_unitPrice = sysComponent_unitPrice + comp_price;
    end
   instance.setValue("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("<stereotype>.<property>")`

- Set a property of an instance: `instance.setValue("<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 must use error handling functions such as `try-catch` statements if it is possible that some elements in the model do not use the stereotypes.

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

```
if instance.isComponent()
  weight = 0;
  for child=instance.Components
    subcomp_weight = child.getValue("PhysicalElement.weight");
    weight = weight + subcomp_weight;
  end
  instance.setValue("PhysicalElement.weight",weight)
end
```

Once the analysis function is complete, add it to the analysis. 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.

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

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

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

See Also

`systemcomposer.analysis.Instance`

More About

- "Define Profiles and Stereotypes" on page 4-2
- "Use Stereotypes and Profiles" on page 4-9

Battery Sizing and Automotive Electrical System Analysis

Overview

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

Structure of the Model

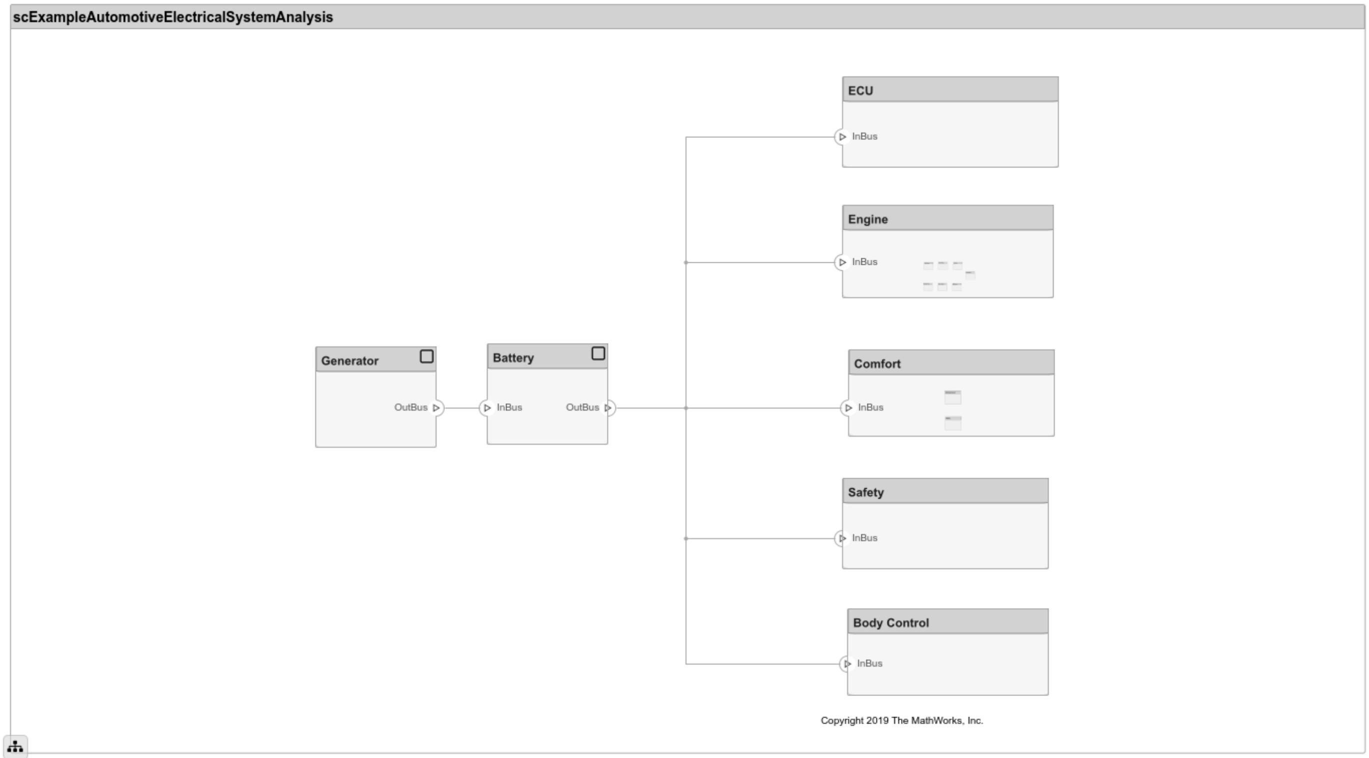
The generator charges the battery while the engine is running. The battery, along with the generator supports the electrical loads in the vehicle, like ECU, radio, and body control. The inductive loads like motors and other coils have the InRushCurrent stereotype property defined. Based on the properties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

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

Load the Model and Run the Analysis

```
archModel = systemcomposer.openModel('scExampleAutomotiveElectricalSystemAnalysis');
% Instantiate battery sizing class used by the analysis function to store
% analysis results.
objcomputeBatterySizing = computeBatterySizing;
% Run the analysis using the iterator.
archModel.iterate('Topdown',@computeLoad,objcomputeBatterySizing);
% Display analysis results.
objcomputeBatterySizing.displayResults;
```

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



Close the Model

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

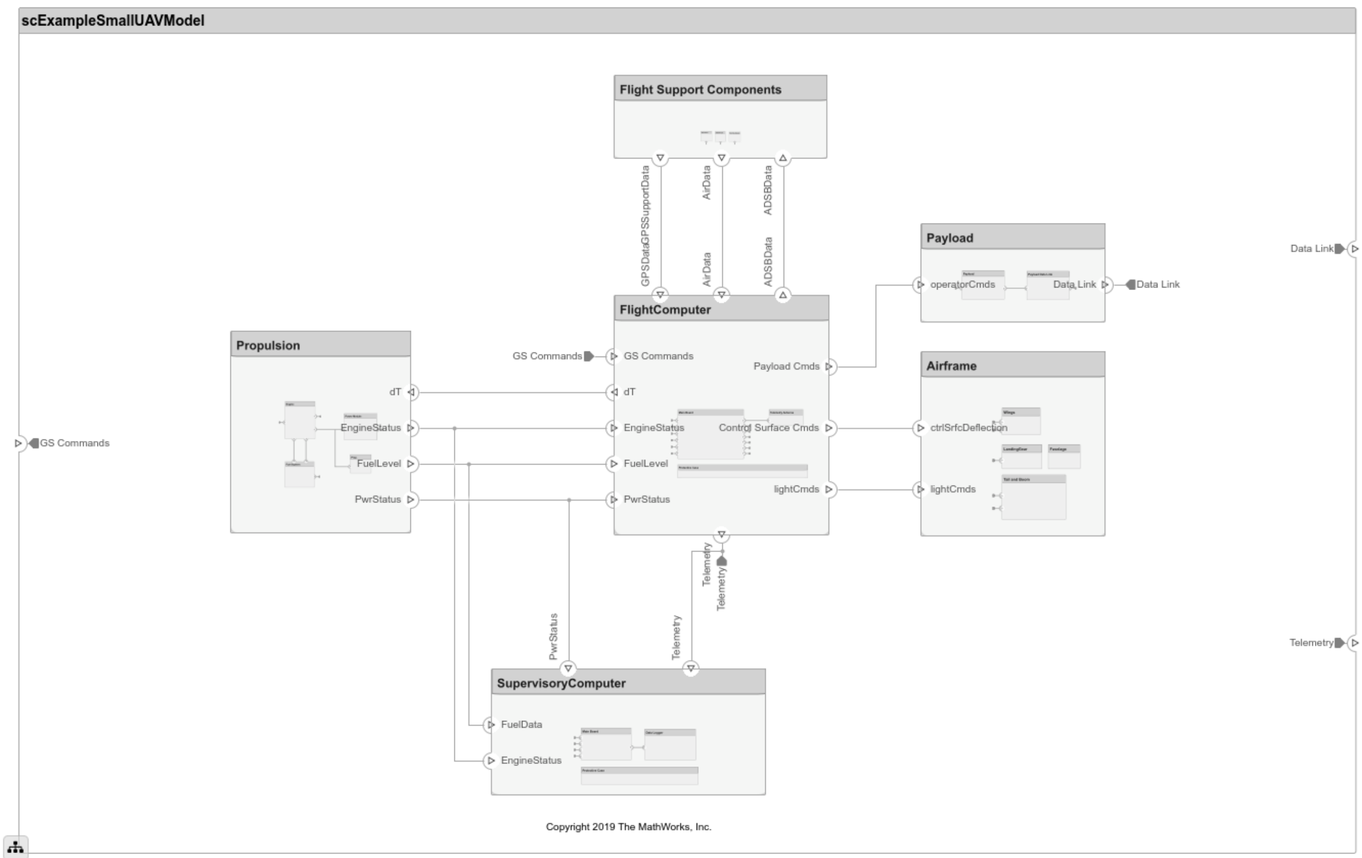
Modeling System Architecture of Small UAV

Overview

This example shows how to set up the architecture for a small unmanned aerial vehicle, composed of six top-level components. You also learn how to refine your architecture design by authorizing interfaces, linking to requirements, defining profiles and stereotypes, and running analysis on such an architecture model.

Open the project.

```
>> scExampleSmallUAV
```

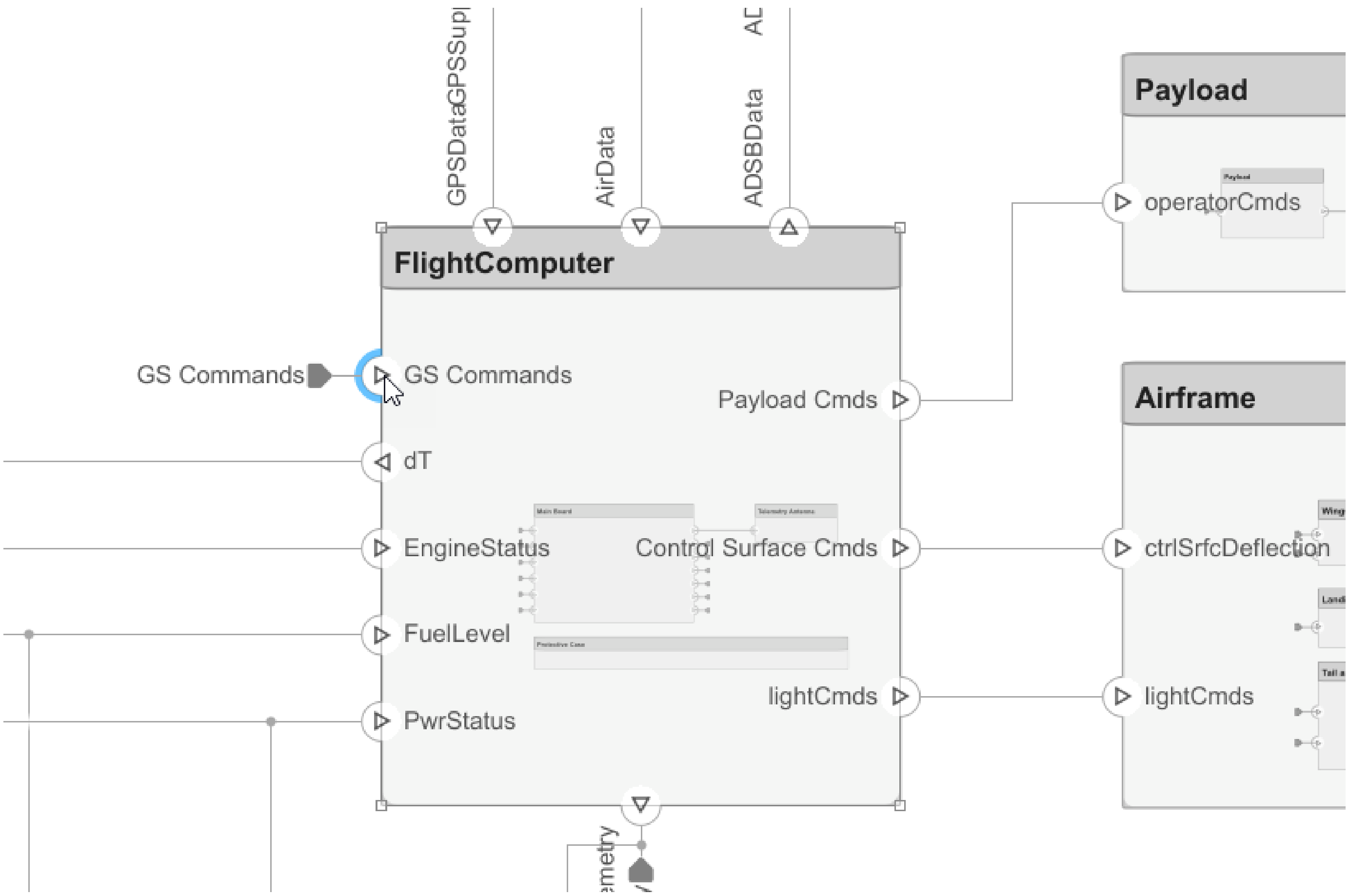


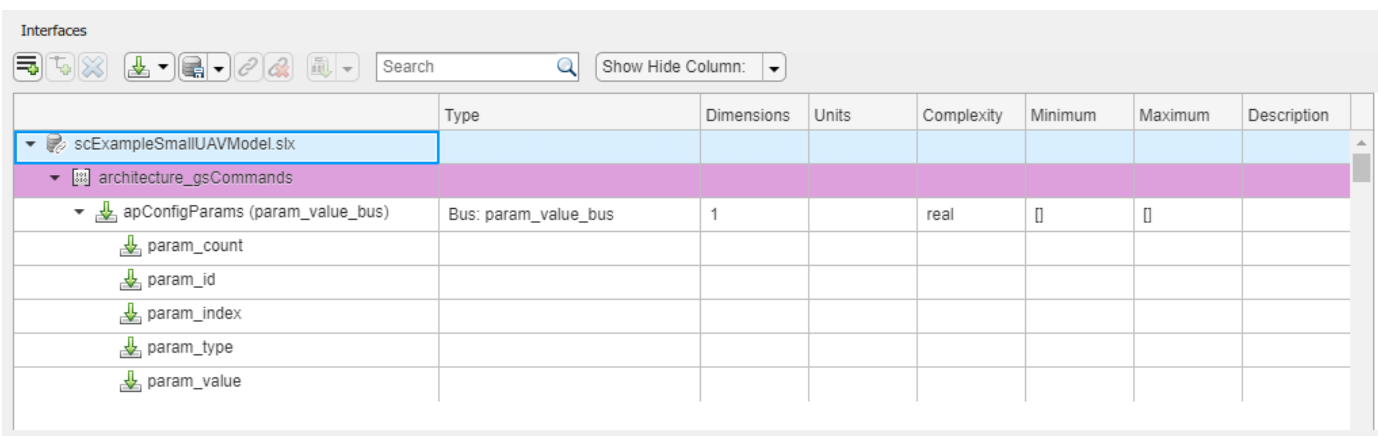
Each top-level component is decomposed into its subcomponents. Navigate through the hierarchy to view the composition for each component. The root component, SmallUAV, has input and output ports that represent data exchange between the system and its environment.

Specify Interfaces

Define interfaces in a data dictionary. From the menu, click on **Interface Editor**.

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

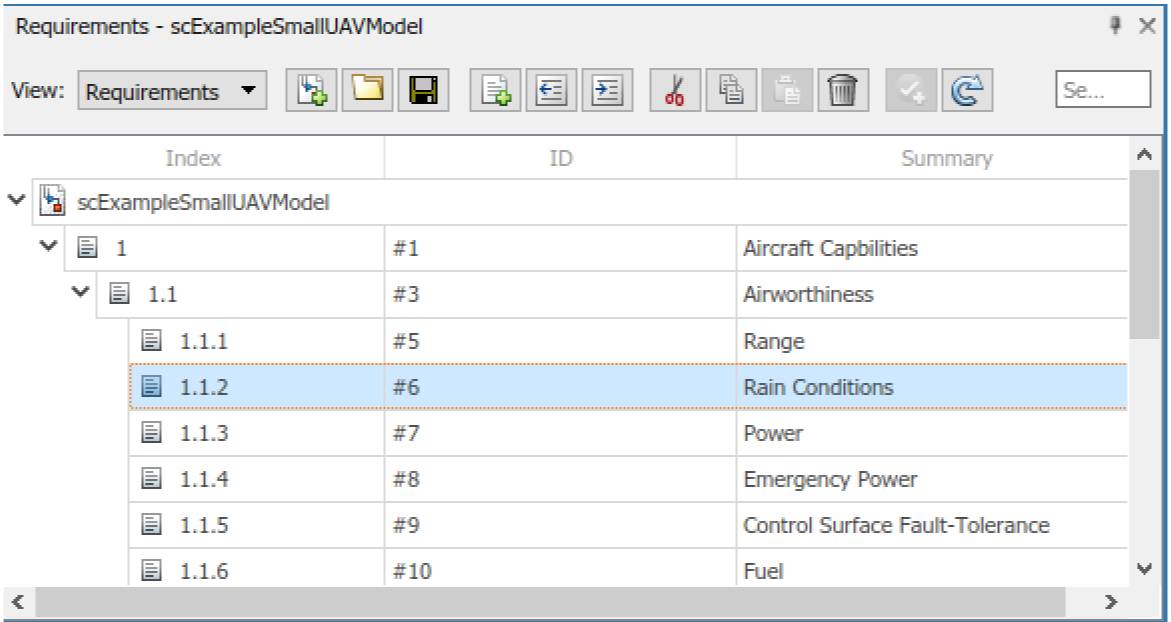




Inspect Requirements

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

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.



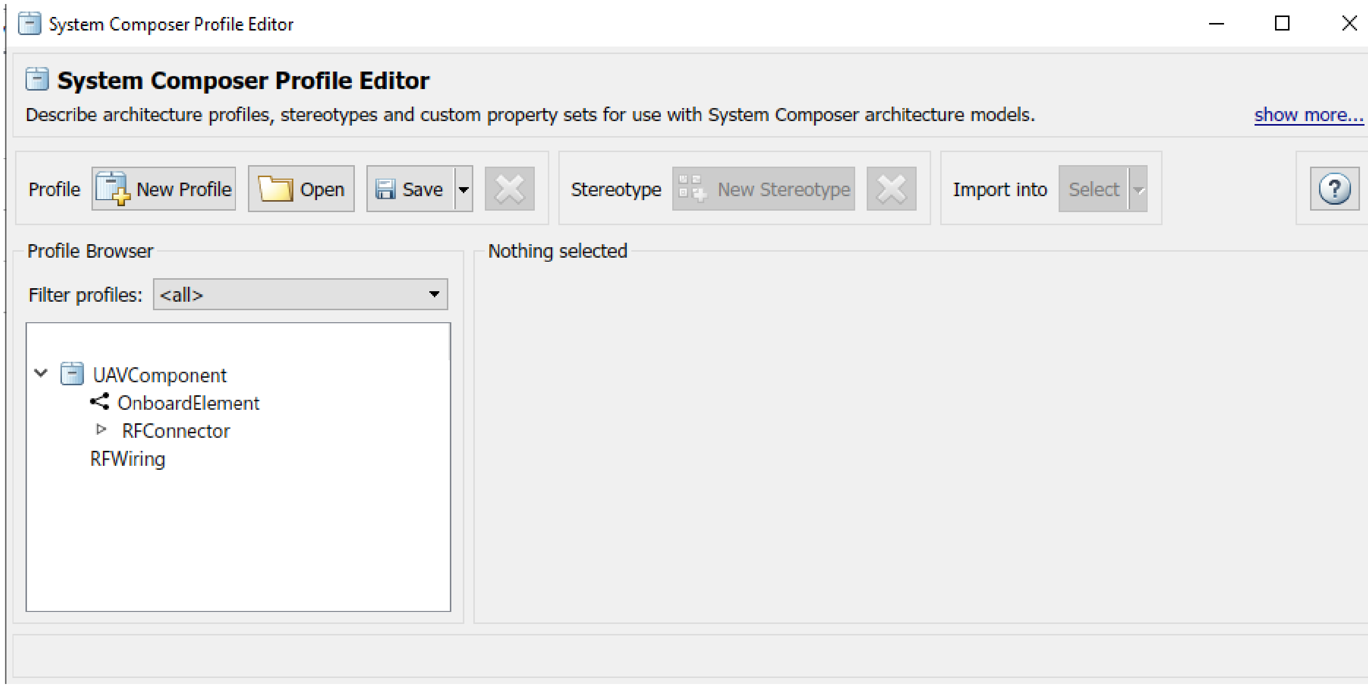
Define Profiles and Stereotypes

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

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

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

The display appears below.

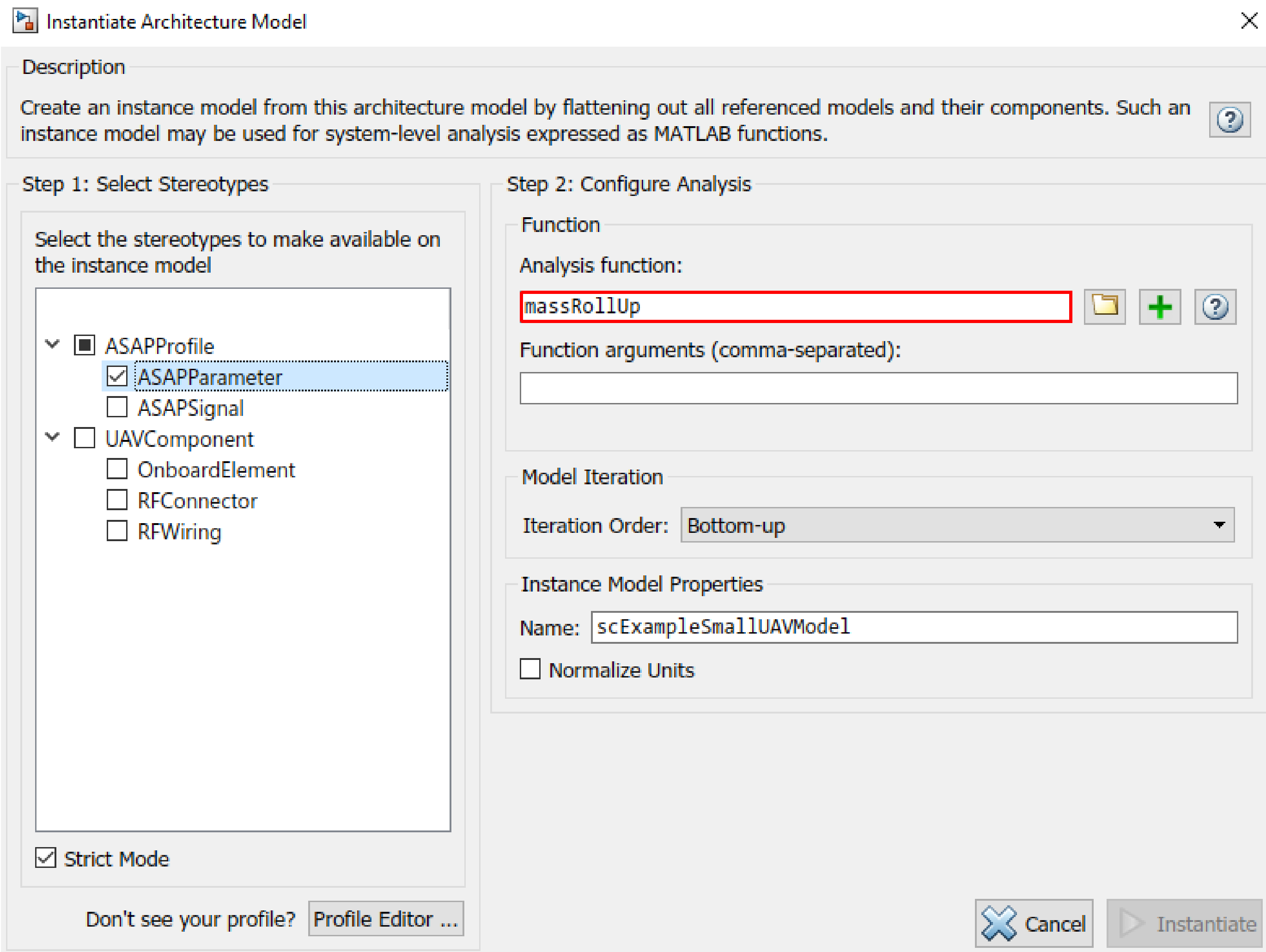


Analyze the Model

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

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

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



Click **Instantiate** to generate an analysis.

Instances	Mass	Power
scExampleSmallUAVModel		
Airframe		
Fuselage	1.7	
LandingGear	1.65	
Tail and Boom	2.7	
Wings	3.2	
Flight Support Components		
ADSB Module		
ABDSB Antenna	0.058	
ADSB Board	0.098	
GPS Module		
GPS Antenna	0.128	
GPS Board	0.27	
Pitot Tube Module	0.075	
FlightComputer		
Main Board	0.145	

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, SmallUAV.

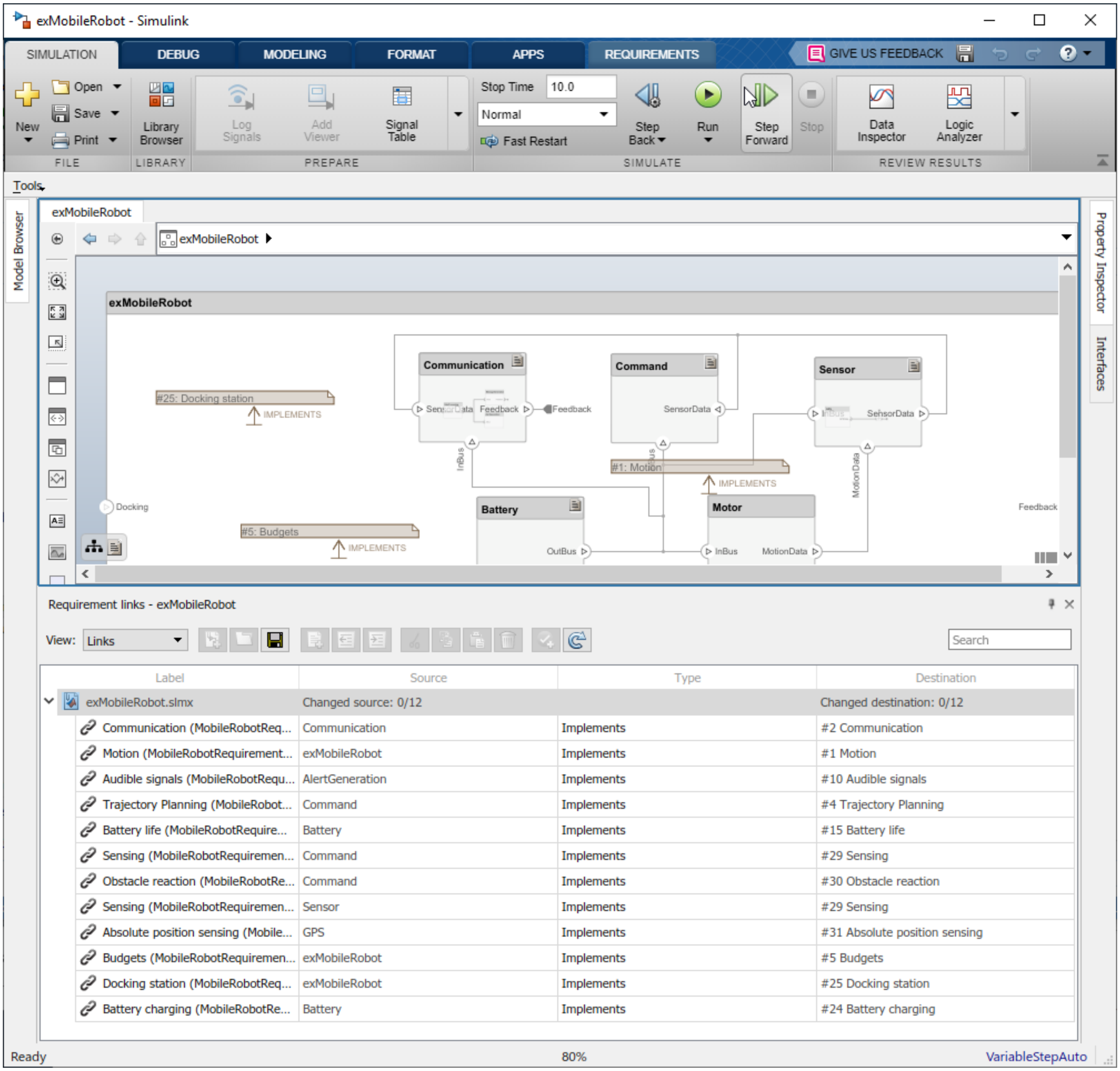
Link and Trace Requirements

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

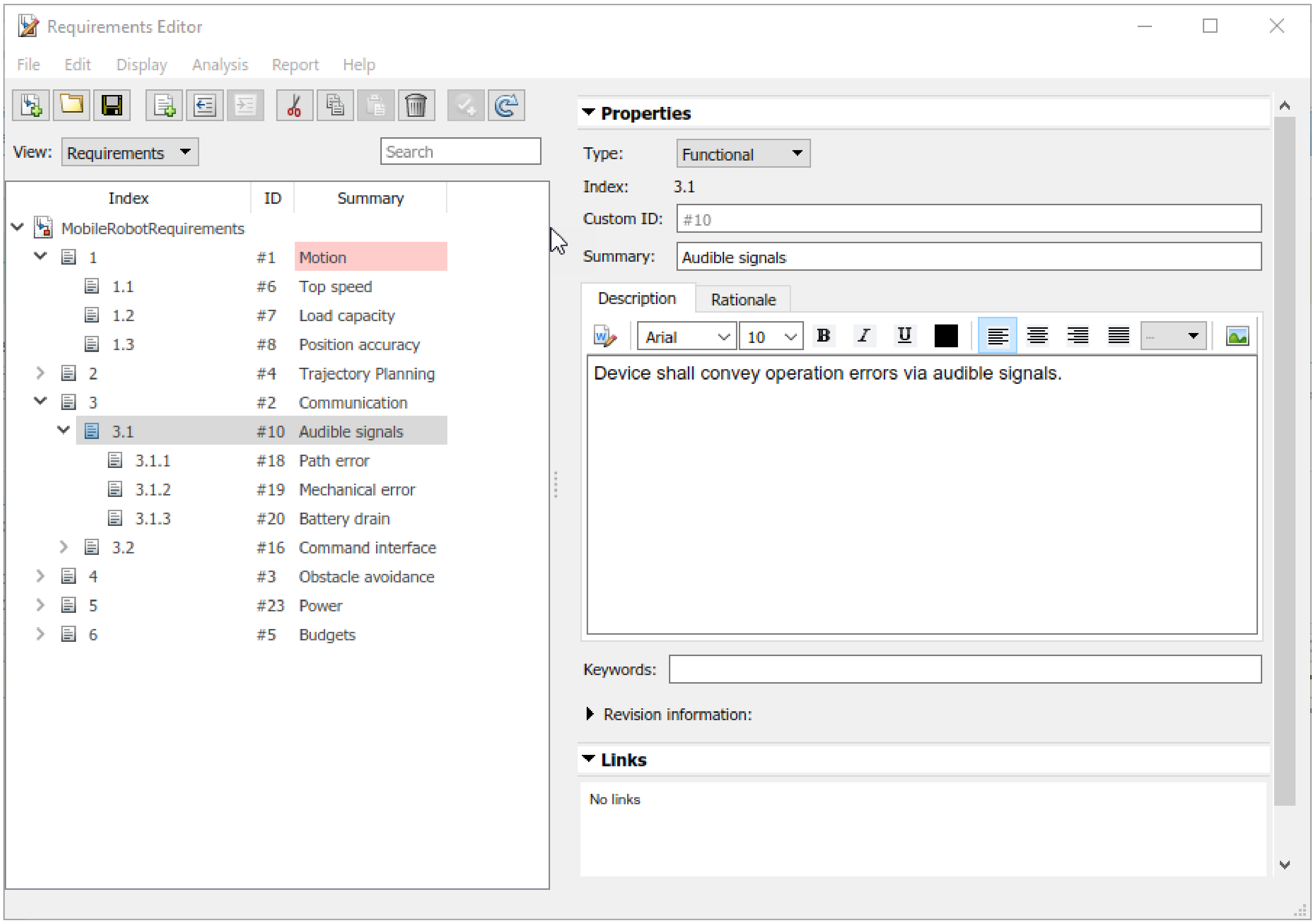
Allocate functional requirements to components to establish traceability. By creating a link between a component and the related requirement, you can track whether all requirements are represented in the architecture. You can also keep requirements and design in sync, for example, if a requirement changes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

Open the model `exMobileRobot`.

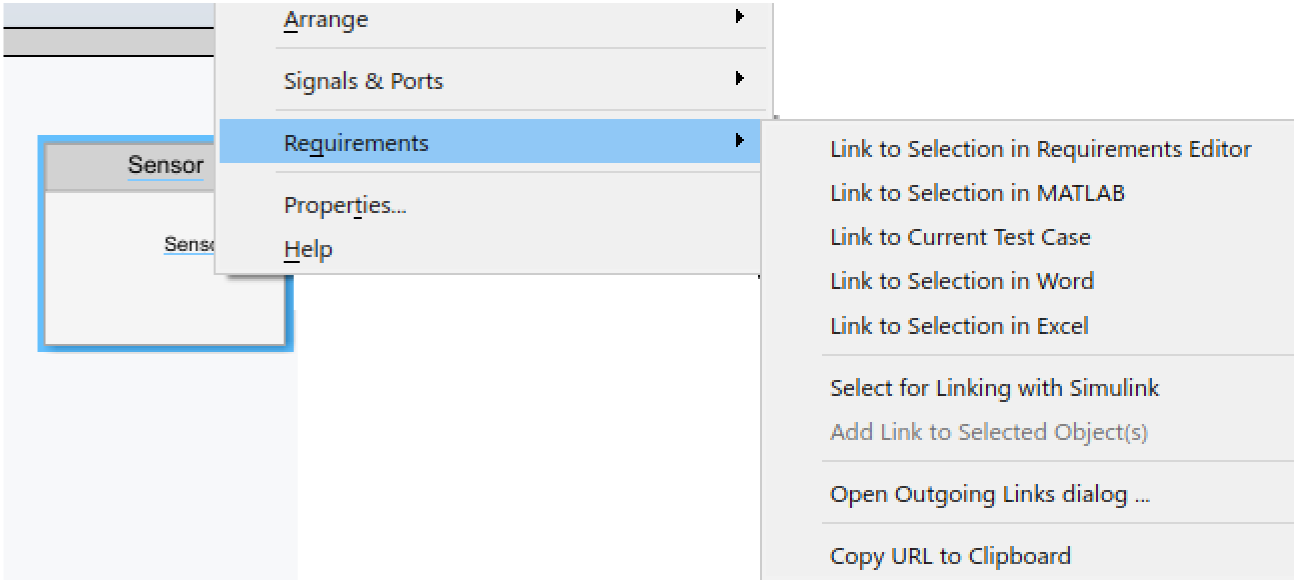
```
open_system('exMobileRobot')
```



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

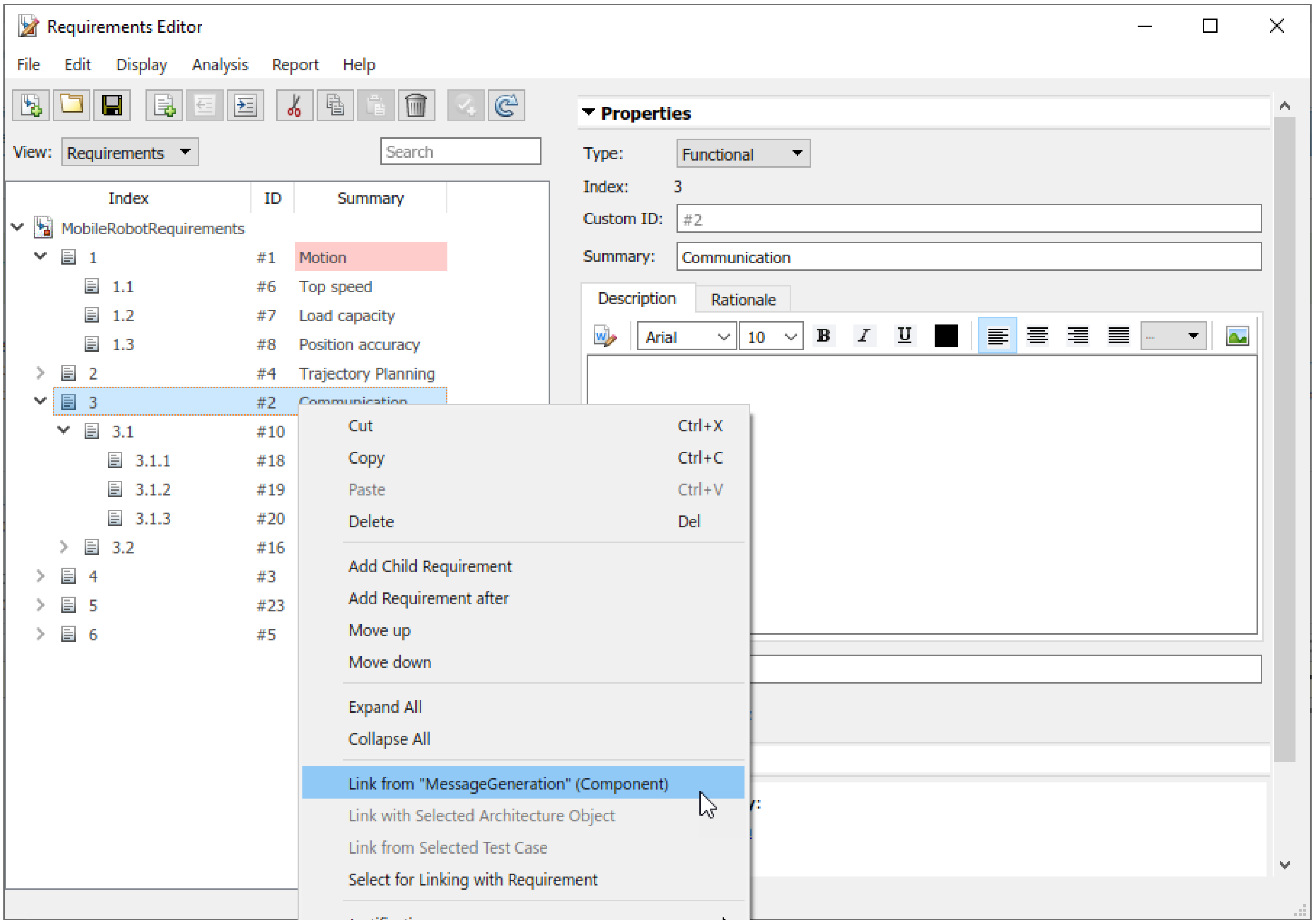


Select the component to be linked in the architecture model. Right-click and select **Requirements > Link to Selection in Requirements Editor**.



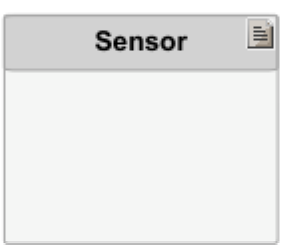
When you first link a requirement in an architecture model, a link set file with extension `.slmx` is created to store requirement links. The **Requirements** context menu displays the linked requirements.

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

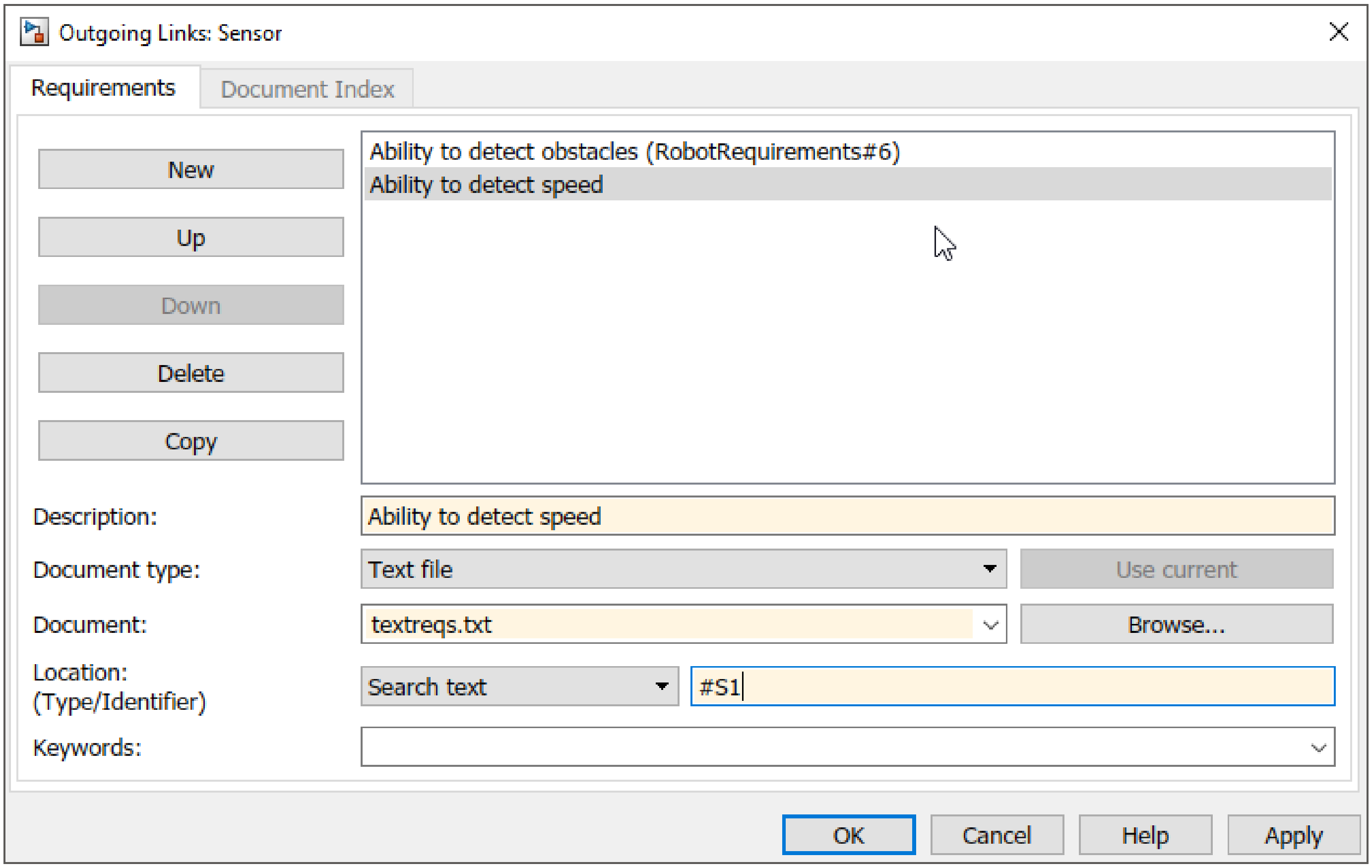


You can also create requirement links with blocks and subsystems in Simulink models. for more information, see “Link Blocks and Requirements” (Simulink Requirements).

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



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



Related Topics

- "Manage Requirements" on page 2-2
- "View Linked Requirements in Models and Blocks"

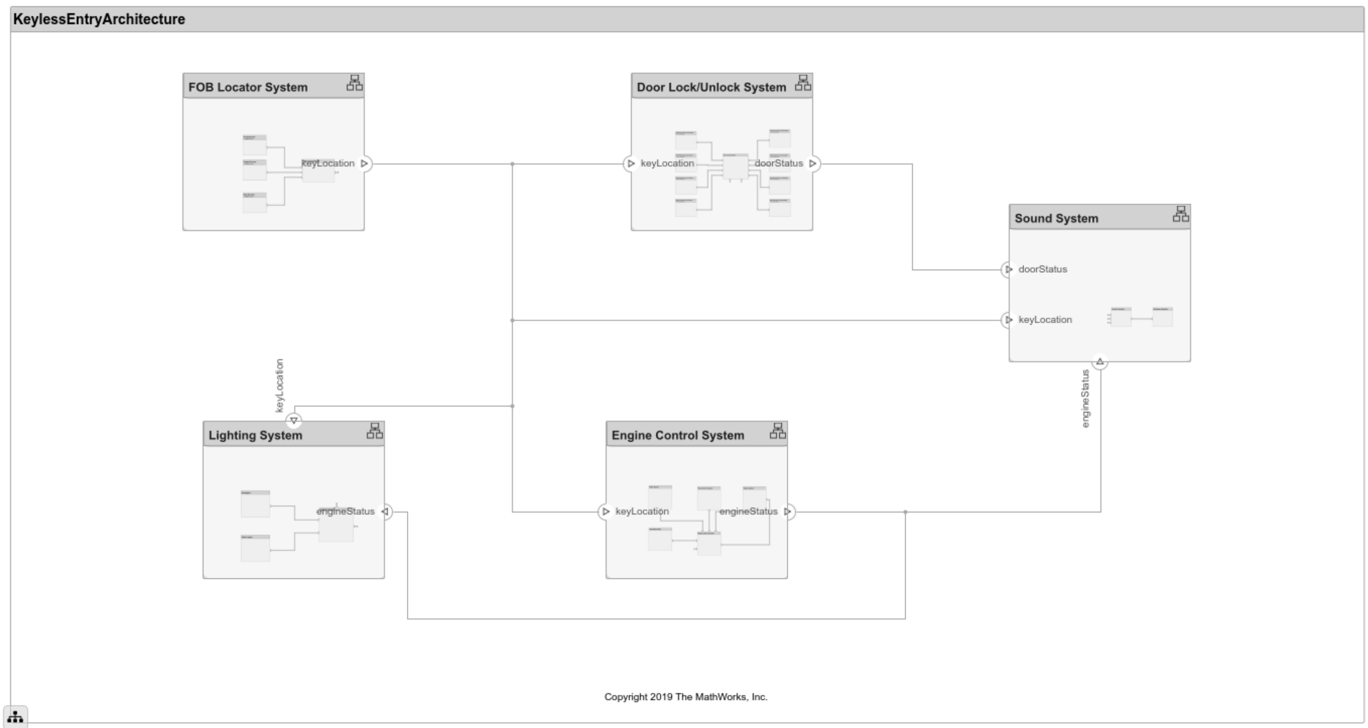
Modeling System Architecture of Keyless Entry System

Overview

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

Open the project.

```
scKeylessEntrySystem
```



Opening the Architecture Views

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

- Key FOB Position Dataflow — A view of the components in the model that are making use of the **KeyFOBPosition** interface.
- Door Lock System Supplier Breakdown — A view of the components in the door lock system grouped by which supplier is providing the given components.
- Sound System Supplier Breakdown — A view of the components in the sound system grouped by which supplier is providing the given components.
- Software Component Review Status — A view of the components in the model with the **SoftwareComponent** stereotype applied grouped by the value of the `ReviewStatus` property.

Extract the Architecture of a Simulink Model Using System Composer

Overview

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

Converting Simulink Model to System Composer Architecture

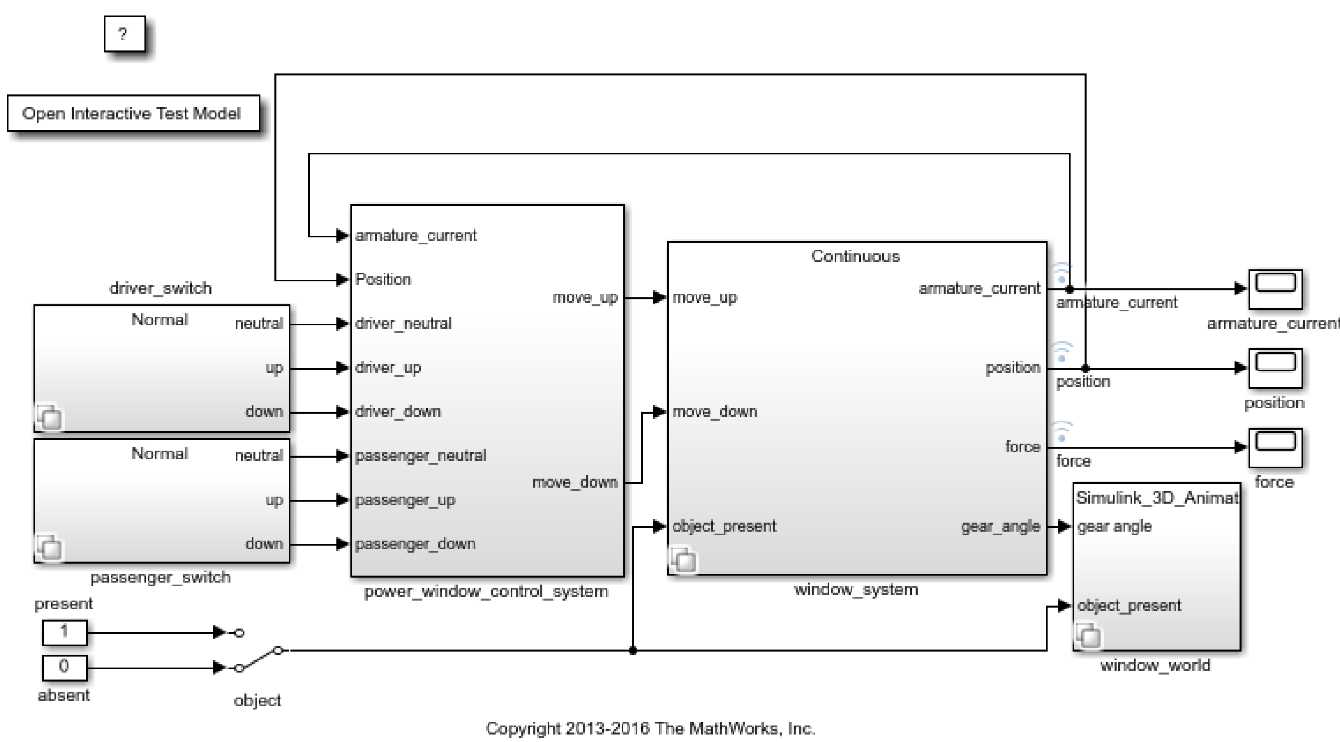
System Composer converts structural constructs in a Simulink model to equivalent architecture model constructs:

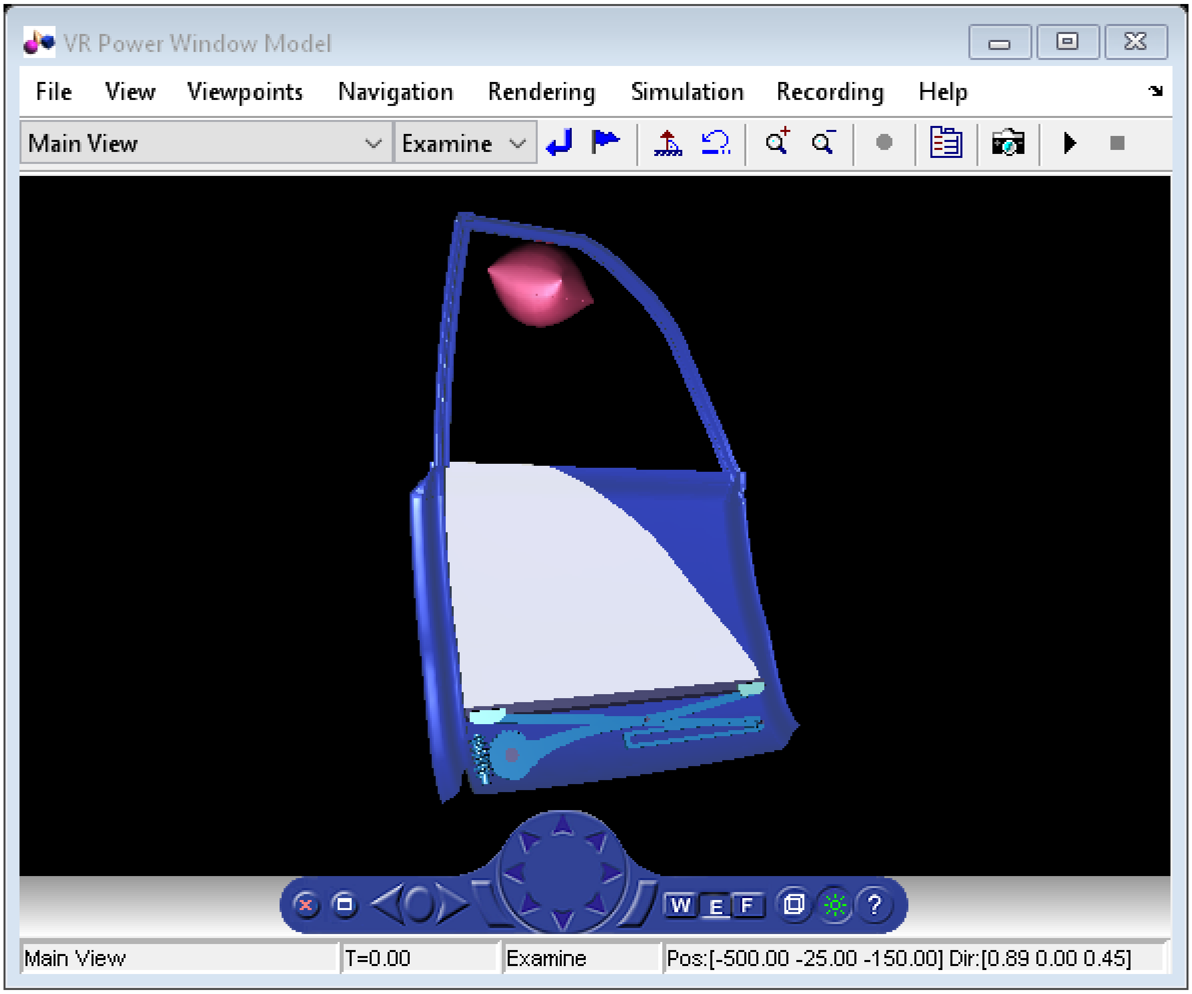
- Subsystems to components
- Variant subsystems to variant components
- Bus objects to interfaces
- Referenced models to reference components

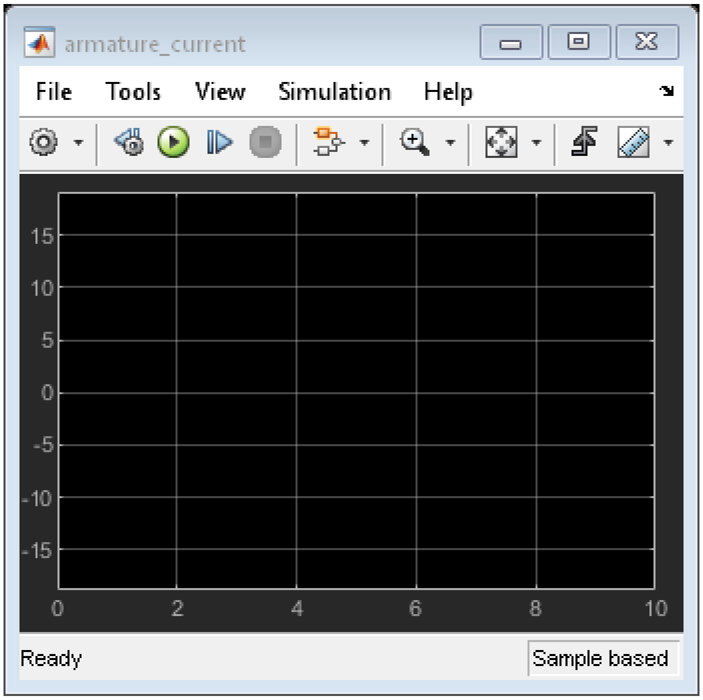
Open the Model

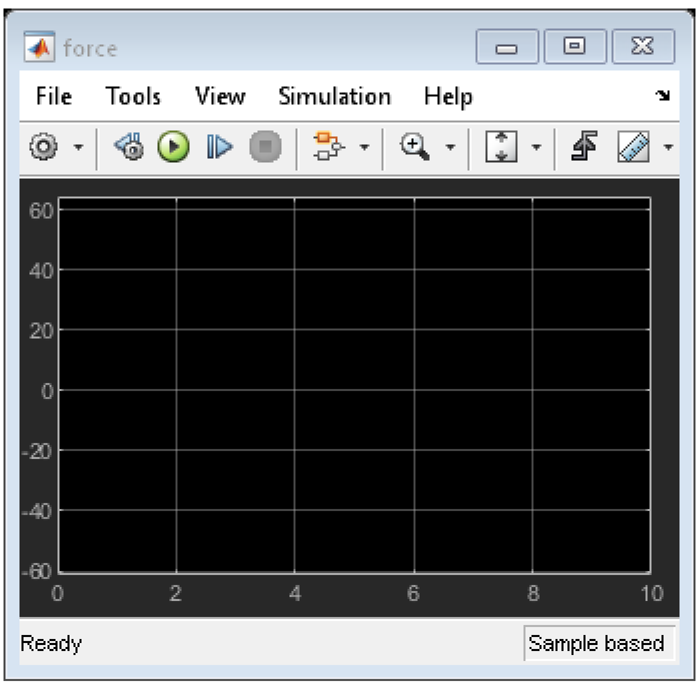
Open the Simulink model of a system.

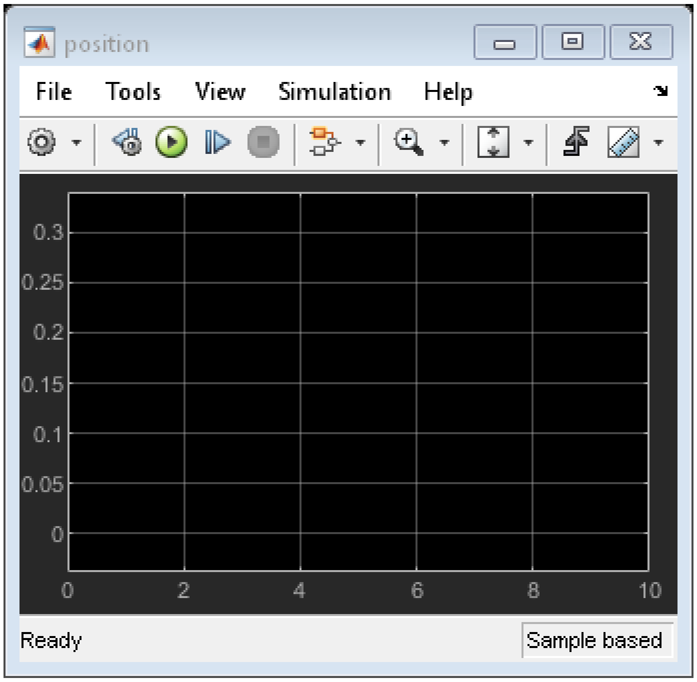
```
scExamplePowerWindowBottomUp
```




VR Power Window Model
File
View
Viewpoints
Navigation
Rendering
Simulation
Recording
Help
Main View
Examine
W E F
Main View
T=0.00
Examine
Pos:[-500.00 -25.00 -150.00] Dir:[0.89 0.00 0.45]


armature_current
File
Tools
View
Simulation
Help
15
10
5
0
-5
-10
-15
0
2
4
6
8
10
Ready
Sample based

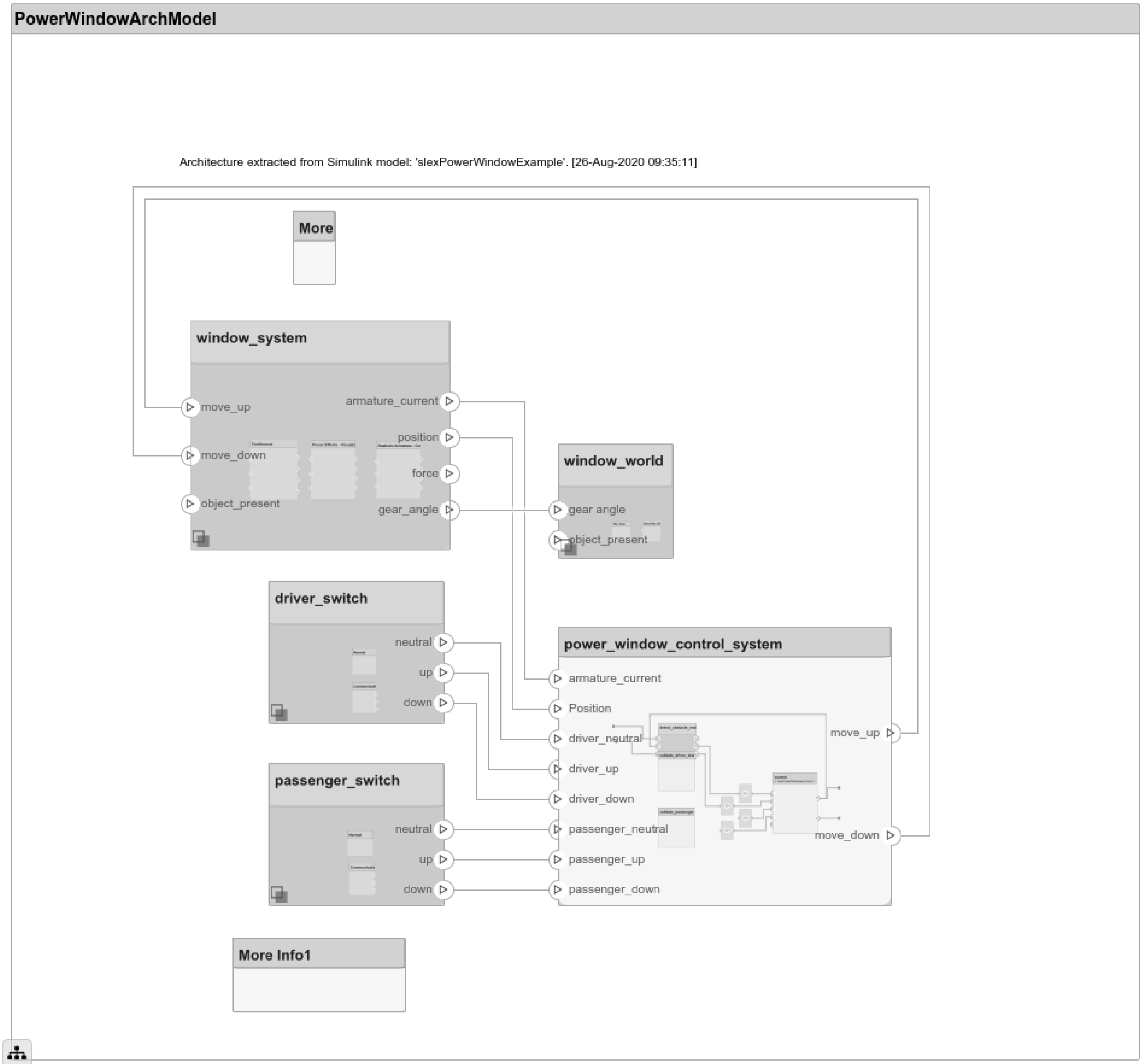

force
File
Tools
View
Simulation
Help
60
40
20
0
-20
-40
-60
0
2
4
6
8
10
Ready
Sample based

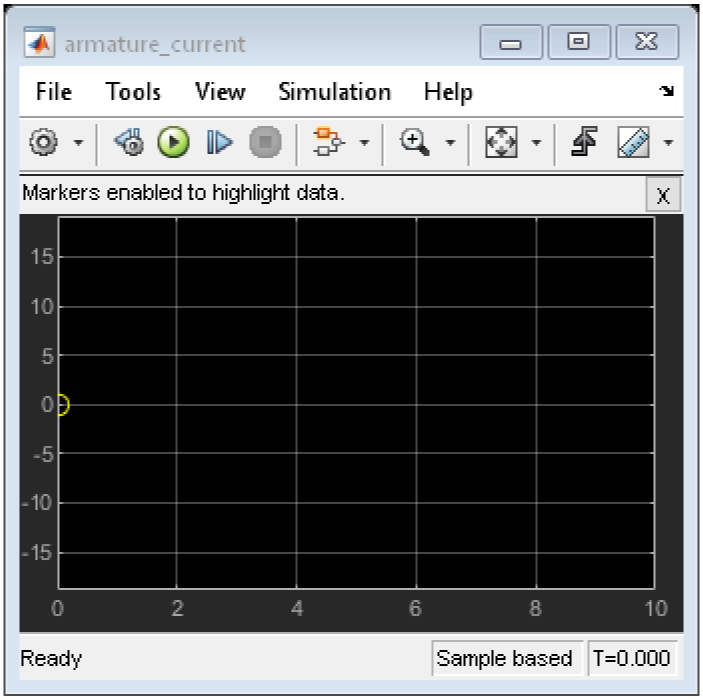


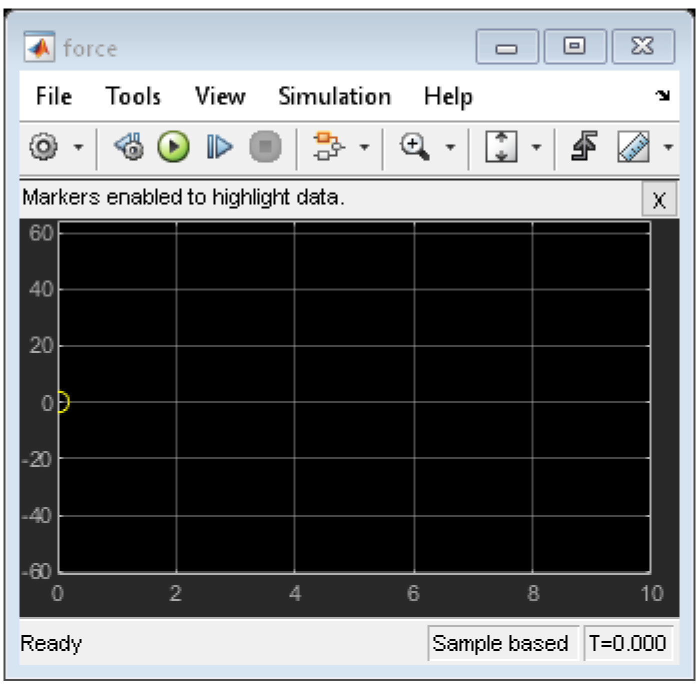
Export the Model

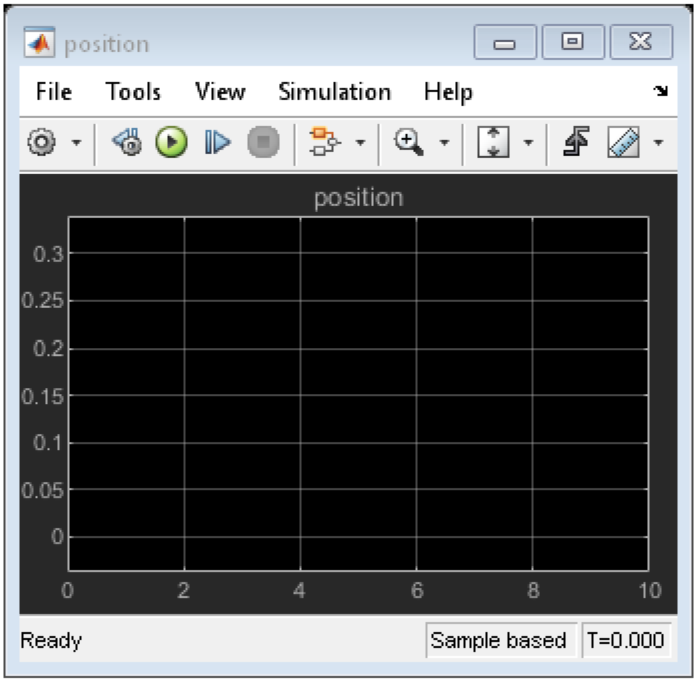
Extract an architecture model from the original model.

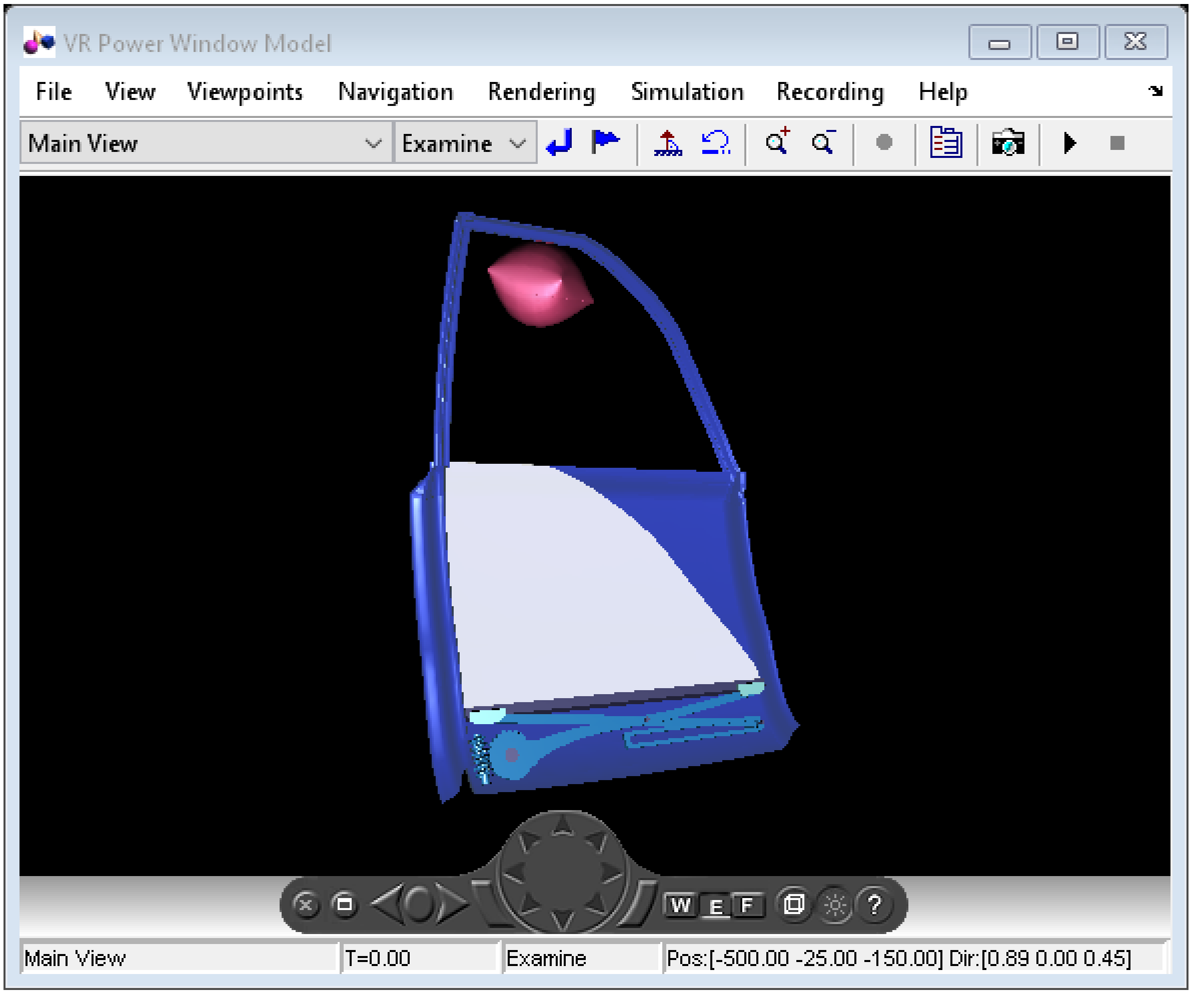
```
systemcomposer.extractArchitectureFromSimulink('slexPowerWindowExample','PowerWindowArchModel');
Simulink.BlockDiagram.arrangeSystem('PowerWindowArchModel');
systemcomposer.openModel('PowerWindowArchModel');
```


PowerWindowArchModel
Architecture extracted from Simulink model: 'slexPowerWindowExample'. [26-Aug-2020 09:35:11]
More
window_system
move_up
move_down
object_present
armature_current
position
force
gear_angle
window_world
gear angle
object_present
driver_switch
neutral
up
down
passenger_switch
neutral
up
down
power_window_control_system
armature_current
Position
driver_neutral
driver_up
driver_down
passenger_neutral
passenger_up
passenger_down
move_up
move_down
More Info1


armature_current
File Tools View Simulation Help
Markers enabled to highlight data.
15
10
5
0
-5
-10
-15
0 2 4 6 8 10
Ready
Sample based
T=0.000


force
File Tools View Simulation Help
Markers enabled to highlight data.
60
40
20
0
-20
-40
-60
0 2 4 6 8 10
Ready
Sample based
T=0.000


position
File
Tools
View
Simulation
Help
position
0.3
0.25
0.2
0.15
0.1
0.05
0
0
2
4
6
8
10
Ready
Sample based
T=0.000



Close the Model and Project

Close the Simulink project and the created architecture model.

```
bdclose('PowerWindowArchModel');
close(proj);
```

Build an Architecture Model from Command Line

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

Prepare Workspace

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

Build a Model

Add Components, Ports, and Connections

```
model = systemcomposer.createModel('mobileRobotAPI');
arch = model.Architecture;
components = addComponent(arch,{'Sensor','Planning','Motion'});
sensorPorts = addPort(components(1).Architecture,{'MotionData','SensorData'},{'in','out'});
planningPorts = addPort(components(2).Architecture,{'Command','SensorData','MotionCommand'},{'in
motionPorts = addPort(components(3).Architecture,{'MotionCommand','MotionData'},{'in','out'});
c_sensorData = connect(arch,components(1),components(2));
c_motionData = connect(arch,components(3),components(1));
c_motionCommand = connect(arch,components(2),components(3));
```

Add and Connect an Architecture Port

Add a port on the architecture. This is an architecture port.

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

The `connect` command requires a component port as argument. Obtain the component port and connect:

```
compPort = getPort(components(2),'Command');
c_Command = connect(archPort,compPort);
```

Save the model.

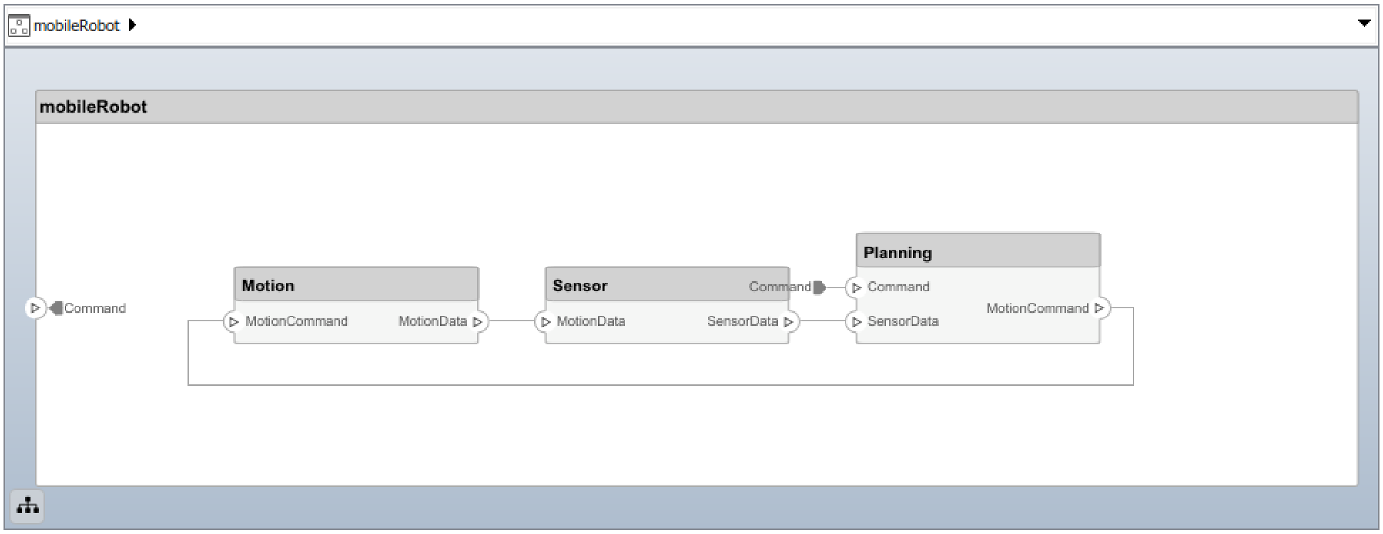
```
save(model)
```

Open the model

```
open_system(gcs);
```

Arrange the layout by pressıng **Ctrl+Shift+A** or using the following command.

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



Create and Apply Profile and Stereotypes

Profiles are `xml` files that can be applied to any model.

Create a Profile and Add Stereotypes

Create a profile.

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

Create a stereotype that applies to all element types:

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

Create stereotypes for different types of components. These types are dictated by design needs and are at the discretion of the user:

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

Create a stereotype for connections:

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

Add Properties

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

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

Apply Profile to Model

Apply profile to the model:

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

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

```
applyStereotype(components(2),'GeneralProfile.softwareComponent')
applyStereotype(components(1),'GeneralProfile.physicalComponent')
applyStereotype(components(3),'GeneralProfile.physicalComponent')
```

Apply the connector stereotype to all connections:

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

Apply the general element stereotype to all connectors and ports:

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

Set properties for each component:

```
setProperty(components(1),'GeneralProfile.projectElement.ID','001');
setProperty(components(1),'GeneralProfile.projectElement.Description','''Central unit for all se
setProperty(components(1),'GeneralProfile.physicalComponent.Cost','200');
setProperty(components(1),'GeneralProfile.physicalComponent.Weight','450');
setProperty(components(2),'GeneralProfile.projectElement.ID','002');
setProperty(components(2),'GeneralProfile.projectElement.Description','''Planning computer''');
setProperty(components(2),'GeneralProfile.softwareComponent.develCost','20000');
setProperty(components(2),'GeneralProfile.softwareComponent.develTime','300');
setProperty(components(3),'GeneralProfile.projectElement.ID','003');
setProperty(components(3),'GeneralProfile.projectElement.Description','''Motor and motor control
setProperty(components(3),'GeneralProfile.physicalComponent.Cost','4500');
setProperty(components(3),'GeneralProfile.physicalComponent.Weight','2500');
```

Set the properties of connections to be identical:

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

Create an Interface

Create a data dictionary and add an interface:

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

Link the interface to the model:

```
linkDictionary(model,'SensorInterfaces.sldd');
```

Identify the interface in the dictionary:

```
interface_GPS = getInterface(model.InterfaceDictionary,'GPSInterface');
```

Set the interface for the port:

```
setInterface(sensorPorts(2),interface_GPS);
```

Save Data Dictionary

Save the changes to the data dictionary.

```
dictionary.save();
```

Clean Up

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

```
% bdclose('mobileRobotAPI');
% systemcomposer.profile.Profile.closeAll;
% delete('SensorInterfaces.sldd');
```

Import and Export Architectures

This example shows how to import and export architectures. In System Composer, an architecture is fully defined by three sets of information:

- Component information
- Port information
- Connection information

You can import an architecture into System Composer when this information is defined in, or converted into, MATLAB tables.

In this example, the architecture information of a simple UAV system is defined in an Excel spreadsheet and is used to create a System Composer architecture model. It also links elements to the specified system level requirement. You can modify the files in this example to import architectures defined in external tools, when the data includes the required information. The example also shows how to export this architecture information from System Composer architecture model to an Excel spreadsheet.

Architecture Definition Data

You can characterize the architecture as a network of components and import by defining components, ports, connections, interfaces and requirement links in MATLAB tables. The component table must include name, unique ID, and parent component ID for each component. It can also include other relevant information required to construct the architecture hierarchy for referenced model, and stereotype qualifier names. The port table must include port name, direction, component, and port ID information. Port interface information may also be required to assign ports to components. The connection table includes information to connect ports. At a minimum, this table must include the connection ID, source port ID, and destination port ID.

The systemcomposer.importModel(importModelName) API :

- Reads stereotype names from Component table and load the profiles
- Creates components and attaches ports
- Creates connections using the connection map
- Sets interfaces on ports
- Links elements to specified requirements
- Saves referenced models
- Saves the architecture model

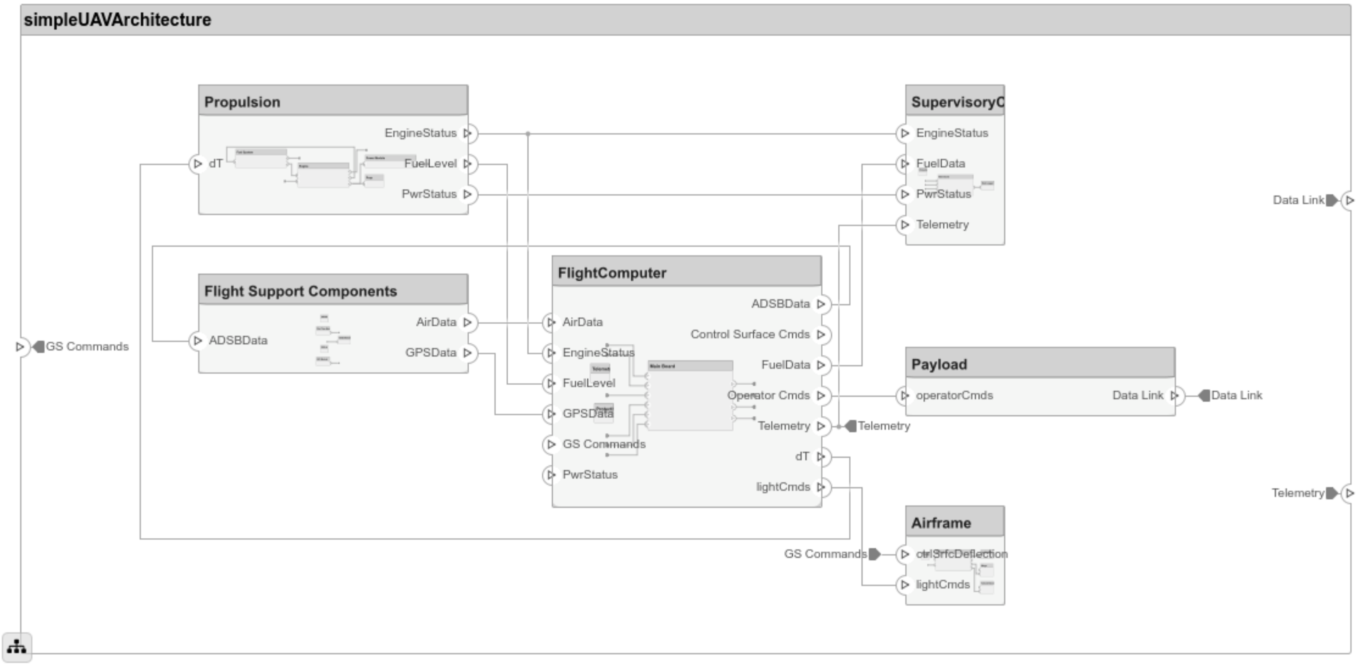
Make sure the current directory is writable because this example will create files.

```
[stat, fa] = fileattrib(pwd);
if ~fa.UserWrite
    disp('This script must be run in a writable directory');
    return;
end
% 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','Por
importAdapter.readTableFromExcel();
```

Import an Architecture

```
model = systemcomposer.importModel(modelName,importAdapter.Components,importAdapter.Ports,import
% Auto-arrange blocks in the generated model
Simulink.BlockDiagram.arrangeSystem(modelName);
```



Export an Architecture

You can export an architecture to MATLAB tables and then convert to an external file

```
exportedSet = systemcomposer.exportModel(modelName);
% The output of the function is a structure that contains the component table, port table,
% connection table, the interface table, and the requirement links table.
% Save the above structure to excel file.
SaveToExcel('ExportedUAVModel',exportedSet);
```

Close Model

```
bdclose(modelName);
```

Import System Composer Architecture using Model Builder.

This example shows how to import architecture specifications into System Composer using the systemcomposer.io.modelBuilder() utility class. These architecture specifications can be defined in external source such as Excel file.

In system composer, an architecture is fully defined by three sets of information:

- Components and its position in architecture hierarchy
- Ports and its mapping to components
- Connections between the components through ports In this example, we also import interface data definitions from external source.
- Interfaces in architecture models and its mapping to ports

This example uses systemcomposer.modelBuilder class to pass all of the above architecture information and import a System Composer model.

In this example, architecture information of a small UAV system is defined in an Excel spreadsheet and is used to create a System Composer architecture model.

External Source Files

- Architecture.xlsx : This Excel file contains hierarchical information of the architecture model. This example maps the external source data to System Composer model elements. Below is the mapping of information in column names to System Composer model elements.

```
# Element    : Name of the element. Either can be component or port name.
# Parent     : Name of the parent element.
# Class      : Can be either component or port(Input/Output direction of the port).
# Domain     : Mapped as component property. Property "Manufacturer" defined in the
               profile UAVComponent under Stereotype PartDescriptor maps to Domain values in
# Kind       : Mapped as component property. Property "ModelName" defined in the
               profile UAVComponent under Stereotype PartDescriptor maps to Kind values in e
# InterfaceName : If class is of port type. InterfaceName maps to name of the interface link
# ConnectedTo : In case of port type, it specifies the connection to
                other port defined in format "ComponentName::PortName".
```

- DataDefinitions.xlsx : This excel file contains interface data definitions of the model. This example assumes the below mapping between the data definitions in the source excel file and interfaces hierarchy in System Composer :

```
# Name        : Name of the interface or element.
# Parent      : Name of the parent interface Name(Applicable only for elements) .
# Datatype    : Datatype of element. Can be another interface in format
                Bus: InterfaceName
# Dimensions  : Dimensions of the element.
# Units       : Unit property of the element.
# Minimum     : Minimum value of the element.
# Maximum     : Maximum value of the element.
```

Step 1. Instantiate the model builder class

You can instantiate the model builder class with a profile name.

Make sure the current directory is writable because this example will be creating files.

```
[stat, fa] = fileattrib(pwd);
if ~fa.UserWrite
    disp('This script must be run in a writable directory');
    return;
end
% Name of the model to build.
modelName = 'scExampleModelBuider';
% Name of the profile.
profile = 'UAVComponent';
% Name of the source file to read architecture information.
architectureFileName = 'Architecture.xlsx';

% Instantiate the ModelBuilder
builder = systemcomposer.io.ModelBuilder(profile);
```

Step 2. Build Interface Data Definitions.

Reading the information in external source file DataDefinitions.xlsx, we build the interface data model.

Create MATLAB tables from source Excel file.

```
opts = detectImportOptions('DataDefinitions.xlsx');
opts.DataRange = 'A2'; % force readtable to start reading from the second row.
definitionContents = readtable('DataDefinitions.xlsx', opts);

% systemcomposer.io.IdService class generates unique ID for a
% given key
idService = systemcomposer.io.IdService();

for rowItr =1:numel(definitionContents(:,1))
    parentInterface = definitionContents.Parent{rowItr};
    if isempty(parentInterface)
        % In case of interfaces adding the interface name to model builder.
        interfaceName = definitionContents.Name{rowItr};
        % Get unique interface ID. getID(container,key) generates
        % or returns(if key is already present) same value for input key
        % within the container.
        interfaceID = idService.getID('interfaces',interfaceName);
        % Builder utility function to add interface to data
        % dictionary.
        builder.addInterface(interfaceName,interfaceID);
    else
        % In case of element read element properties and add the element to
        % parent interface.
        elementName  = definitionContents.Name{rowItr};
        interfaceID = idService.getID('interfaces',parentInterface);
        % ElementID is unique within a interface.
        % Appending 'E' at start of ID for uniformity. The generated ID for
        % input element is unique within parent interface name as container.
        elemID = idService.getID(parentInterface,elementName,'E');
        % Datatype, dimensions, units, minimum and maximum properties of
        % element.
        datatype = definitionContents.DataType{rowItr};
        dimensions = string(definitionContents.Dimensions(rowItr));
        units = definitionContents.Units(rowItr);
        % Make sure that input to builder utility function is always a
        % string.
```

```
        if ~ischar(units)
            units = '';
        end
        minimum = definitionContents.Minimum{rowItr};
        maximum = definitionContents.Maximum{rowItr};
        % Builder function to add element with properties in interface.
        builder.addElementInInterface(elementName, elemID, interfaceID, datatype, dimensions, uni
    end
end
```

Step 3. Build Architecture Specifications.

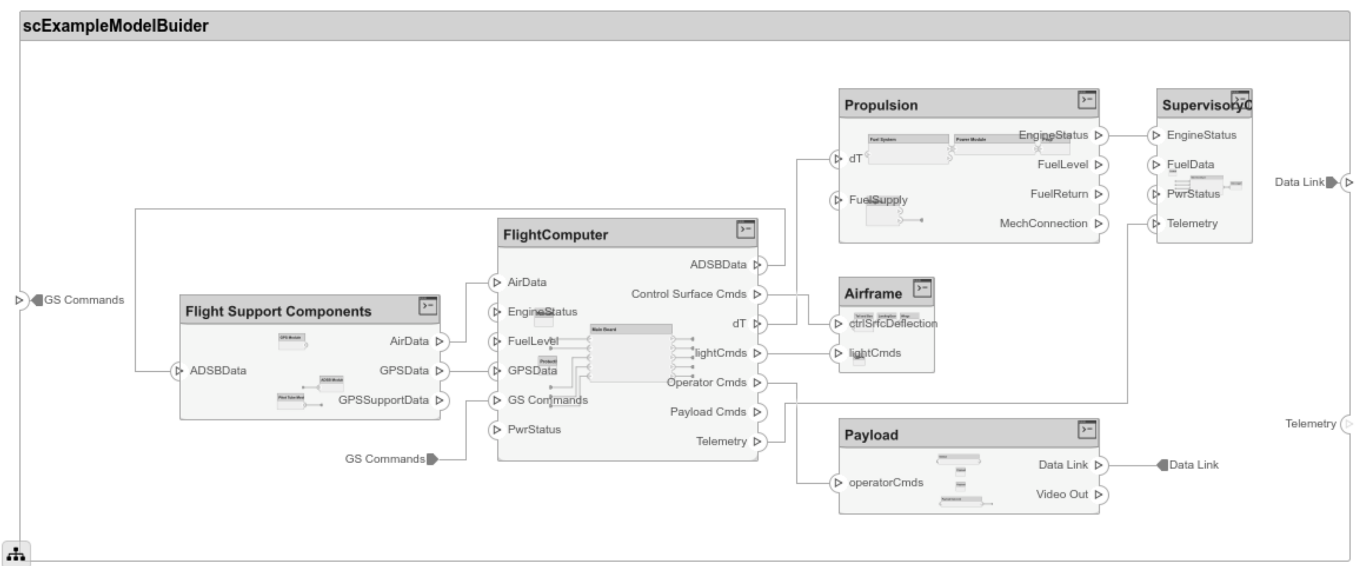
Architecture specifications de Create MATLAB tables from source Excel file.

```
excelContents = readtable(architectureFileName);
% Iterate over each row in table.
for rowItr =1:numel(excelContents(:,1))
% Read each row of the excel file and columns.
    class = excelContents.Class(rowItr);
    Parent = excelContents.Parent(rowItr);
    Name = excelContents.Element{rowItr};
    % Populating the contents of table using the builder.
    if strcmp(class,'component')
        ID = idService.getID('comp',Name);
        % Root ID is by default set as zero.
        if strcmp(Parent,'scExampleSmallUAV')
            parentID = "0";
        else
            parentID = idService.getID('comp', Parent);
        end
        % Builder utility function to add component.
        builder.addComponent(Name,ID,parentID);
        % Reading the property values
        kind = excelContents.Kind{rowItr};
        domain = excelContents.Domain{rowItr};
        % *Builder to set stereotype and property values*
        builder.setComponentProperty(ID, 'StereotypeName','UAVComponent.PartDescriptor','ModelNam
    else
        % In this example, concatenation of port name and parent component name
        % is used as key to generate unique IDs for ports.
        portID = idService.getID('port',strcat(Name,Parent));
        % For ports on root architecture. compID is assumed as "0".
        if strcmp(Parent,'scExampleSmallUAV')
            compID = "0";
        else
            compID = idService.getID('comp',Parent);
        end
        % Builder utility function to add port.
        builder.addPort(Name, class, portID, compID );

        % InterfaceName specifies the name of the interface linked to port.
        interfaceName = excelContents.InterfaceName{rowItr};

        % Get interface ID. getID() will return the same IDs already
        % generated while adding interface in Step 2.
        interfaceID = idService.getID('interfaces',interfaceName);
        % Builder to map interface to port.
        builder.addInterfaceToPort(interfaceID, portID);
```

```
            % Reading the connectedTo information to build connections between
            % components.
            connectedTo = excelContents.ConnectedTo{rowItr};
            % connectedTo is in format -:
            % (DestinationComponentName::DestinationPortName).
            % For this example, considering the current port as source of the connection.
            if ~isempty(connectedTo)
                connID = idService.getID('connection',connectedTo);
                splits = split(connectedTo,'::');
                % Get the port ID of the connected port.
                % In this example, port ID is generated by concatenating
                % port name and parent component name. If port id is already
                % generated getID() function returns the same id for input key.
                connectedPortID = idService.getID('port',strcat(splits(2),splits(1)));
                % Using builder to populate connection table.
                sourcePortID = portID;
                destPortID = connectedPortID;
                % Builder to add connections.
                builder.addConnection(connectedTo,connID,sourcePortID,destPortID);
            end
        end
    end
```

Step 3. Builder build method imports model from populated tables.

```
[model, importReport] = builder.build(modelName);
```



Close Model

```
bdclose(modelName);
```