Simulink® Test™

User's Guide

MATLAB®&SIMULINK®

R2019a

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

Simulink® Test™ User's Guide

Revision History

March 2015	Online only	New for Version 1.0 (Release 2015a)
September 2015	Online only	Revised for Version 1.1 (Release 2015b)
October 2015	Online only	Rereleased for Version 1.0.1 (Release 2015aSP1)
March 2016	Online only	Revised for Version 2.0 (Release 2016a)
September 2016	Online only	Revised for Version 2.1 (Release 2016b)
March 2017	Online Only	Revised for Version 2.2 (Release 2017a)
September 2017	Online Only	Revised for Version 2.3 (Release 2017b)
March 2018	Online Only	Revised for Version 2.4 (Release 2018a)
September 2018	Online Only	Revised for Version 2.5 (Release 2018b)
March 2019	Online Only	Revised for Version 3.0 (Release 2019a)

Contents

3 Test Sequences and Assessments

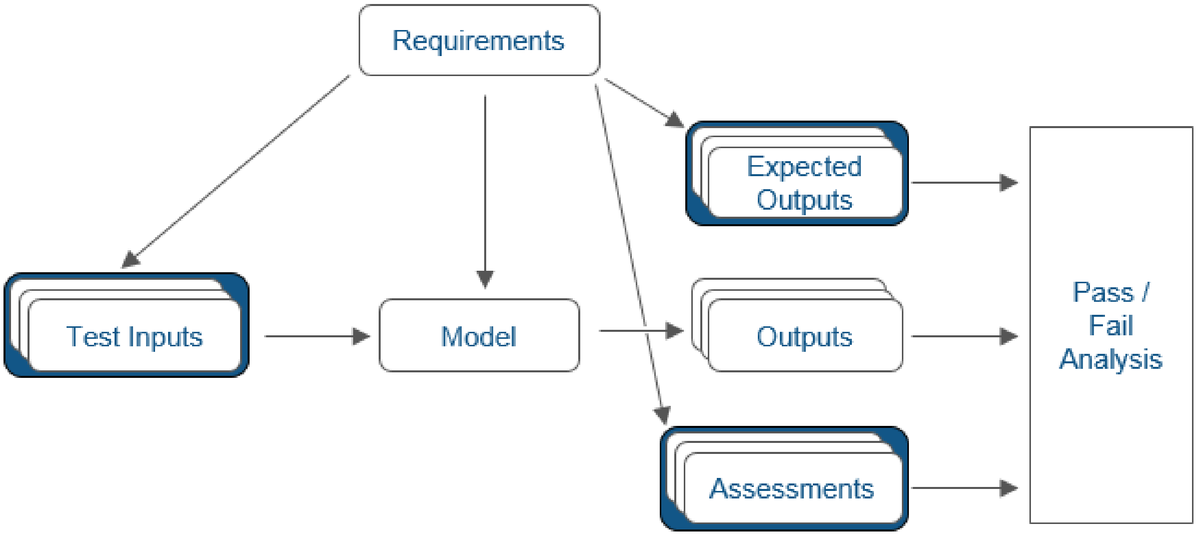
1

Test Strategies

Link to Requirements

In this section...
"Requirements Traceability Considerations" on page 1-2
"Establish Requirements Traceability for Testing" on page 1-3

Since requirements specify behavior in response to particular conditions, you can develop test inputs, expected outputs, and assessments from the model requirements.



Requirements Traceability Considerations

Consider the following limitations working with requirements links in test harnesses:

- Some blocks and subsystems are recreated during test harness rebuild operations. Requirements linking is not supported for these blocks and subsystems in a test harness:
  - Conversion subsystems between the component under test and the sources or sinks
  - Test Sequence blocks that schedule function calls

  - Blocks that drive control input signals to the component under test
  - Blocks that drive Goto or From blocks that pass component under test signals
  - Data Store Read and Data Store Write blocks
- If you use external requirements storage, performing the following operations requires reestablishing requirements links to model objects inside test harnesses:
  - Cut/paste or copy/paste a subsystem with a test harness
  - Clone a test harness
  - Move a test harness from a linked block to the library block

Establish Requirements Traceability for Testing

If you have a Simulink Test and a Simulink Requirements™ license, you can link requirements to test harnesses, test sequences, and test cases. Before adding links, review "Supported Requirements Document Types" (Simulink Requirements).

Requirements Traceability for Test Harnesses

When you edit requirements links to the component under test, the links immediately synchronize between the test harness and the main model. Other changes to the component under test, such as adding a block, synchronize when you close the test harness. If you add a block to the component under test, close and reopen the harness to update the main model before adding a requirement link.

To view items with requirements links, select **Analysis** > **Requirements** > **Highlight Model**.

Requirements Traceability for Test Sequences

In test sequences, you can link to test steps. To create a link, first find the model item, test case, or location in the document you want to link to. Right-click the test step, select **Requirements**, and add a link or open the link editor.

To highlight or unhighlight test steps that have requirements links, toggle the requirements links highlighting button in the Test Sequence Editor toolstrip. Highlighting test steps also highlights the model block diagram.

Requirements Traceability for Test Cases

If you use many test cases with a single test harness, link to each specific test case to distinguish which blocks and test steps apply to it. To link test steps or test harness blocks to test cases,

1. Open the test case in the Test Manager.
2. Highlight the test case in the test browser.
3. Right-click the block or test step, and select **Requirements** > **Link to Current Test Case**.

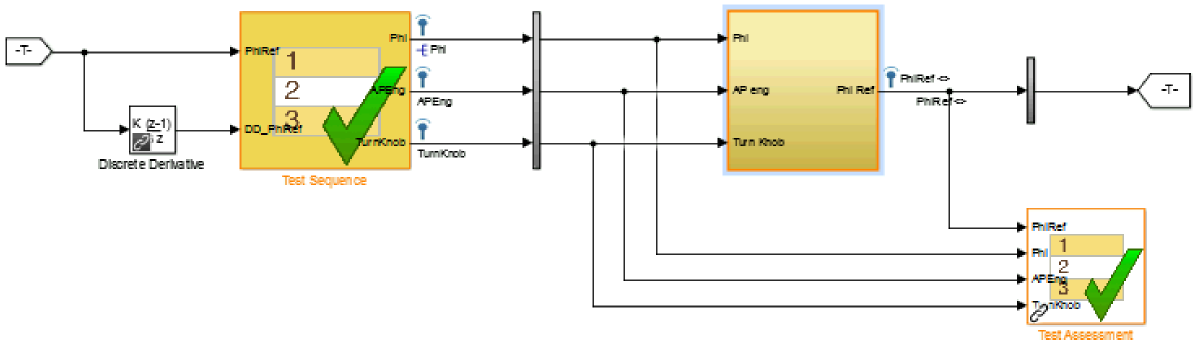
Requirements Traceability Example

This example demonstrates adding requirements links to a test harness and test sequence. The model is a component of an autopilot roll control system. This example requires Simulink Test and Simulink Requirements.

1. Open the test file, the model, and the harness.

   ```
   open AutopilotTestFile.mldatx,
   open_system RollAutopilotMdlRef,
   sltest.harness.open('RollAutopilotMdlRef/Roll Reference',...
   'RollReference_Requirement1_3')
   ```
2. In the test harness, select **Analysis** > **Requirements** > **Highlight Model**.

   The test harness highlights the Test Sequence block, component under test, and Test Assessment block.



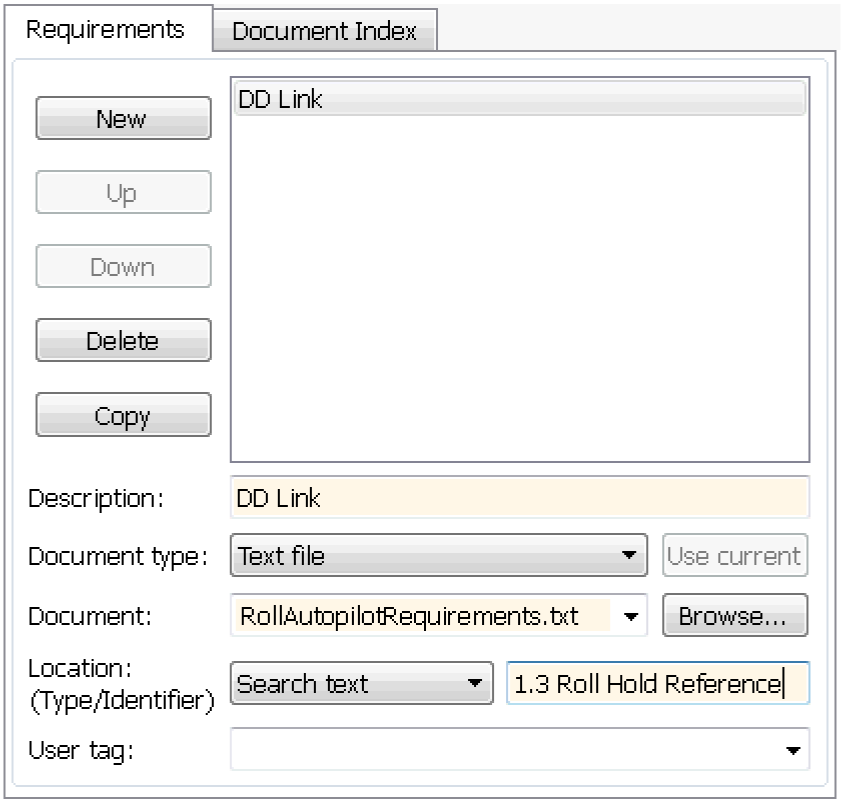
3 Add traceability to the Discrete Derivative block.

a Right-click the Discrete Derivative block and select **Requirements** > **Open Outgoing Links dialog**.

b In the **Requirements** tab, click **New**.

c Enter the following to establish the link:

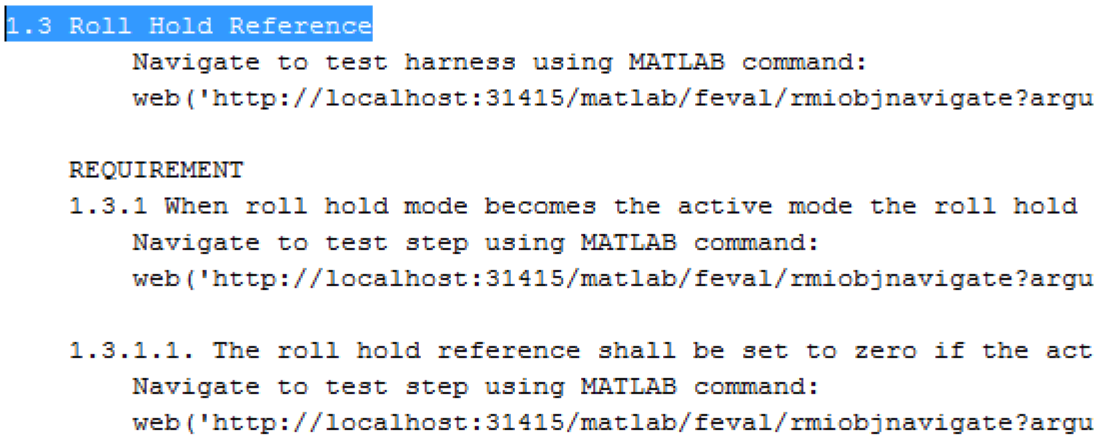
- Description: `DD link`
- Document type: `Text file`
- Document: `RollAutopilotRequirements.txt`
- Location: `1.3 Roll Hold Reference`



Description: DD Link
Document type: Text file
Document: RollAutopilotRequirements.txt
Location: (Type/Identifier) Search text 1.3 Roll Hold Reference
User tag:

d Click **OK**. The Discrete Derivative block highlights.

4 To trace to the requirements document, right-click the Discrete Derivative block, and select **Requirements** > **DD Link**. The requirements document opens in the editor and highlights the linked text.



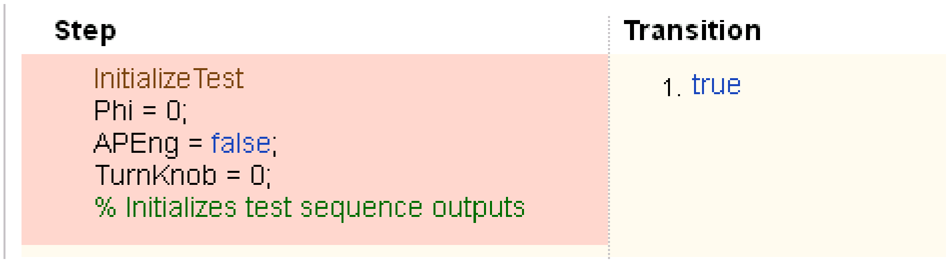
```
1.3 Roll Hold Reference
     Navigate to test harness using MATLAB command:
     web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu

 REQUIREMENT
 1.3.1 When roll hold mode becomes the active mode the roll hold
     Navigate to test step using MATLAB command:
     web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu

 1.3.1.1. The roll hold reference shall be set to zero if the act
     Navigate to test step using MATLAB command:
     web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu
```

5 Open the Test Sequence block. Add a requirements link that links the `InitializeTest` step to the test case.

a In the Test Manager, highlight `Requirement 1.3 Test` in the test browser.

b Right-click the `InitializeTest` step in the Test Sequence Editor. Select **Requirements** > **Link to Current Test Case**.

When the requirements link is added, the Test Sequence Editor highlights the step.



Step	Transition

InitializeTest
Phi = 0;
APEng = false;
TurnKnob = 0;
% Initializes test sequence outputs
	1. true

See Also

Related Examples

- “Requirements-Based Testing for Model Development”

Test Harness

Test Harness and Model Relationship

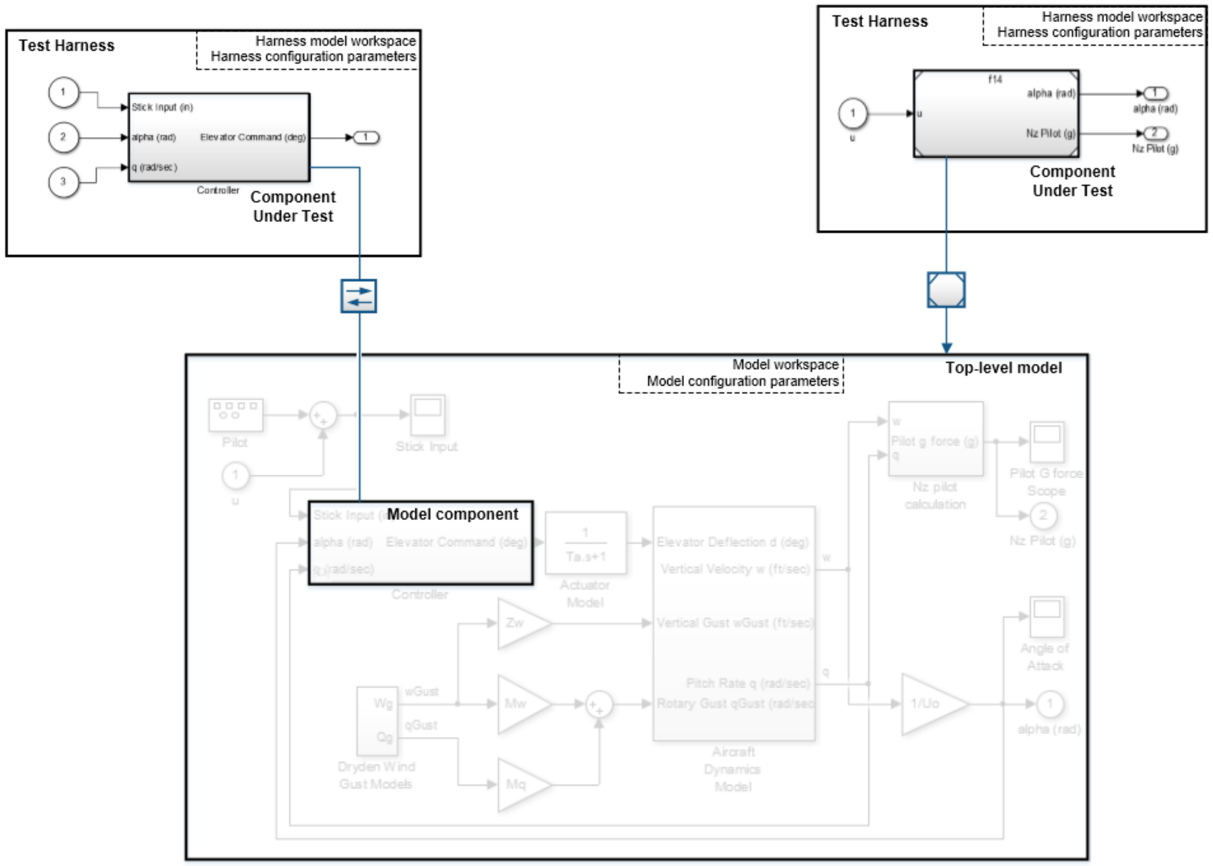
In this section...

Test Harness Description

A test harness is a model block diagram that you can use to test, edit, or debug a Simulink model. In the main model, you associate a harness with a model component or the top-level model. The test harness contains a separate model workspace and configuration set. The test harness is associated with the main model and can be accessed via the model canvas.

You build the test harness model around the component under test, which links the harness to the main model. If you edit the component under test in the harness, the main model updates when you close the harness. You can generate a test harness for:

- A model component, such as a subsystem, library block, or Model block. The test harness isolates the component in a separate simulation environment.
- A top-level model. The component under test is a Model block referencing the main model.

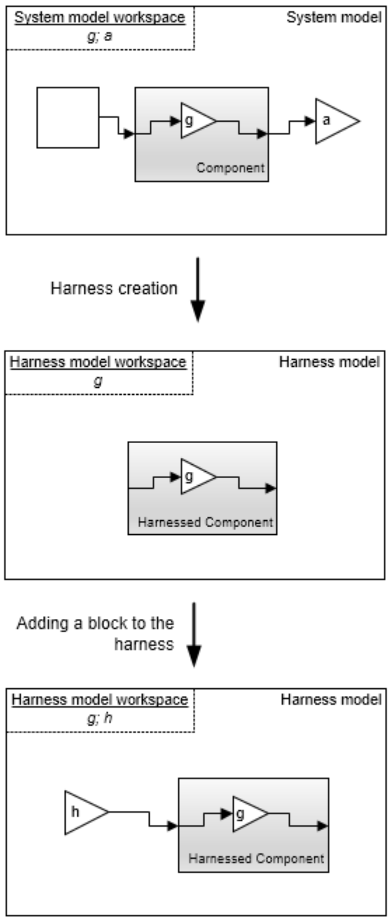


Harness — Model Relationship for a Model Component

When you associate a test harness with a model component, the harness model workspace contains copies of parameters associated with the component.

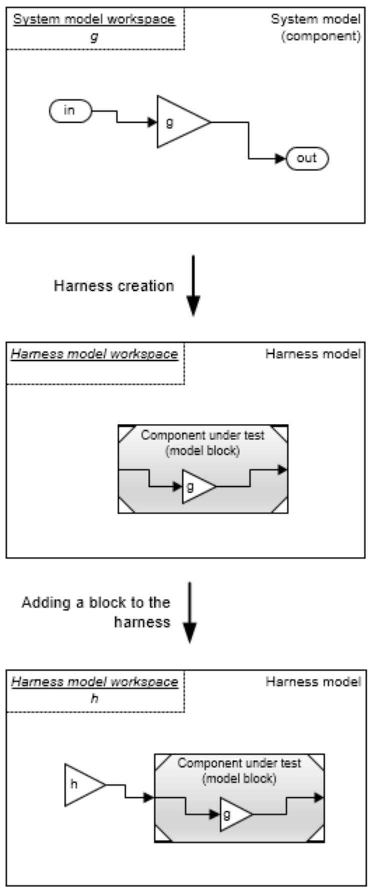
This example shows a test harness for a component that contains a Gain block. The harness model workspace contains a copy of the parameter *g* because *g* defines a part of the component.

The parameter *h* is the gain of a gain block in the harness, outside the component under test (CUT). *h* exists only in the harness model workspace.



Harness — Model Relationship for a Top-Level Model

When you associate a harness with the top level of the main model, the harness model workspace does not contain copies of parameters relevant to the component. The component under test is a Model block referencing the main model, and parameters remain in the main model workspace. In this example, the component under test references the main model, and the variable *g* exists in the main model workspace. The variable *h* is the value of the Gain block in the harness. It exists only in the harness model workspace.



Resolving Parameters

Parameters in the test harness resolve to the most local workspace. Parameters resolve to the harness model workspace, then the system model workspace, then the base MATLAB® workspace.

Test Harness Considerations

- You can open only one test harness at a time per main model.

- Do not comment out the component under test in the test harness. Commenting out the component under test can cause unexpected behavior.
- If a subsystem has a test harness, you cannot expand the subsystem contents into the model containing the subsystem. Delete the test harness(es) before expanding the subsystem. For more information see “Subsystem Expansion” (Simulink).
- Test harnesses are not supported for blocks underneath a Stateflow® object.
- Upgrade advisor and XML differencing are not supported for test harness models.
- A test harness with a Signal Builder block source does not support:
  - Frame-based signals
  - Complex signals
  - Variable-dimension signals
- For a test harness with a Test Sequence block source, all inputs to the component under test must operate with the same sample time.

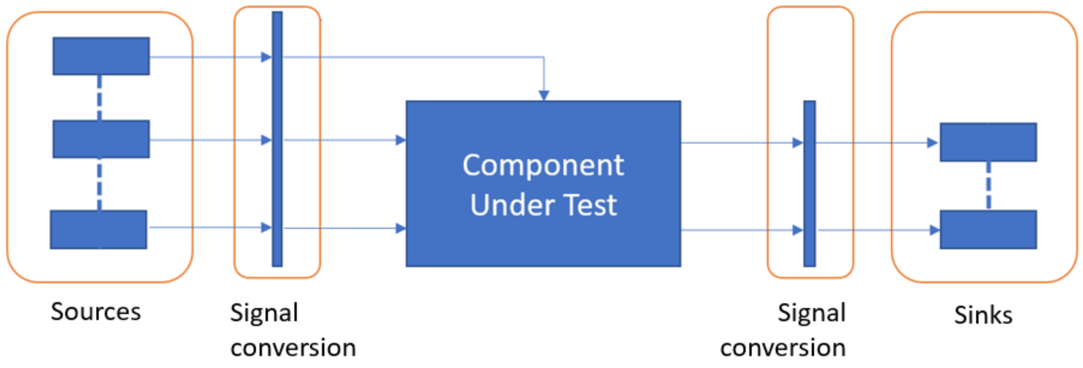
See Also

More About

- “Capabilities of Model Components” (Simulink)

Test Harness Construction for Specific Model Elements

A test harness consists of one or more source blocks that drive the component under test, which drives one or more sink blocks. Test harness construction configures signal attributes, function calls, data stores, and execution semantics. When possible, the test harness matches signal attributes at the sources, sinks, and component interface. For more information on selecting sources and sinks, see “Sources and Sinks” on page 2-16.



Signal Conversion

Signal conversion subsystems adapt the signal interface of the source and sink blocks to the graphical interface of the component. The graphical interface of the component includes input signals, output signals, and action, trigger, or enable inputs. The test harness compiles the main model to determine signal attributes:

- Data type
- Dimensions
- Complexity

Signal attributes are adapted to the sources during harness construction in one of two ways:

1. Source blocks that can generate signals with the compiled attributes are configured to do so.
2. If a source block cannot generate signals with the compiled attributes, signal attribute blocks in the signal conversion subsystem adapt the output of the source

blocks. Signal attribute blocks include Reshape, Rate Transition and Data Type Conversion blocks.

By default, signal conversion subsystems are locked from editing.

Function Calls

Function Call Drivers

If the component under test has function call inputs, a Test Sequence block source generates function call inputs to the component, even if you select a different source during harness creation. To override this behavior and connect function call inputs to your selected source type, create the test harness with the `sltest.harness.create` function, and set `'DriveFcnCallWithTestSequence'` to `false`. For example:

```
sltest.harness.create('Model/FcnCallSubsystem','Source','From File',...
’DriveFcnCallWithTestSequence',false)
```

Function Call Outputs

Function call outputs of the component under test connect to Terminator blocks.

Physical Signal Connections

Components that accept or output physical signals are supported during harness construction, but sources and sinks are not generated. You can add physical modeling blocks to the test harness after construction.

Bus Signals

Test harnesses configuration for bus inputs and outputs depends on the bus connection ability of the source or sink blocks:

1. Sources and sinks that can accept a bus signal are directly connected to the component without modification.
2. If a source cannot output a bus signal, bus signals are automatically constructed from individual bus elements in the signal conversion subsystem.
3. If a sink cannot accept a bus signal, bus signal elements are expanded from the bus signal in the signal conversion subsystem.

String Signals

If the component under test uses string data inputs, and your test harness source does not support string data, string inputs are connected to Ground blocks.

String Inputs

Harness Source Selection	Source Block for String Inputs
Inport	Inport
Signal Builder	Ground
Signal Editor	Ground
From Workspace	Ground
From File	Ground
Test Sequence	Ground
Constant	String Constant (individual string input) Ground (bus containing string)
Ground	Ground

If the component under test uses string data outputs, and your test harness sink does not support string data, string outputs are connected to Terminator blocks.

String Outputs

Harness Sink Selection	Sink Block for String Outputs
Outport	Outport
Scope	Terminator
To Workspace	Terminator
To File	Terminator
Terminator	Terminator

Non-Graphical Connections

In addition to the graphical interface of a component, Simulink supports several non-graphical connections. Test harness construction also supports non-graphical connections.

Goto–From Connections

Goto-From block pairs that cross the component boundary are considered component inputs or outputs.

- A From block without a corresponding Goto block in the component is considered a component input signal. The test harness includes a source block with a corresponding Goto block.
- A Goto block without the corresponding From block in the component is considered a component output signal. The test harness includes a sink block with a corresponding From block.

Data Store Memory

Data Store Read and Data Store Write blocks require a complete data store definition in the test harness.

- If a Data Store Read or Data Store Write block lacks a corresponding Data Store Memory block in the component, the test harness adds a Data Store Memory block.
- For a component containing only Data Store Read blocks, the test harness adds a source block driving a Data Store Write block.
- For a component containing only Data Store Write blocks, the test harness adds a Data Store Read block driving a sink block.

If global data store memory read or write usage cannot be determined, then Data Store Read and Data Store Write blocks are not included in the test harness.

Simulink Function Definitions

If the component calls a Simulink Function that is not defined in the component, the test harness adds a stub Simulink Function block matching the function call signature.

Export Function Models

Test harnesses contain a function-call scheduler for components that use the export-function modeling style. The scheduler is a Test Sequence block that contains prototype calls to the functions in your model.

The scheduler Test Sequence block includes a test step containing:

- A catalog of globally scoped Simulink Function blocks in the component.
- A list of function-call triggers accessible at the component interface.

Harness construction honors periodic function-call triggers with appropriate decimation of the function-call event in the Test Sequence block.

Test harnesses include `Initialize`, `Terminate`, and `Reset` steps for models that contain `Initialize`, `Terminate`, and `Reset` event subsystems. You can include `Initialize`, `Terminate`, and `Reset` steps for other export-function models using the `'ScheduleInitTermReset'` property of `sltest.harness.create`.

Execution Semantics

The execution behavior of a component depends on factors such as computed sample times, solver settings, model configuration, and parameter settings. Execution behavior also depends on run-time events such as function-call triggers and asynchronous events. To handle these execution semantics, test harness construction:

1. Copies configuration parameter settings from the main model into the test harness.
2. Copies required parameter definitions from the main model workspace into the test harness model workspace.
3. Copies data dictionary settings from the main model into the test harness.
4. Honors a limited subset of sample time settings using explicit source block specifications and Rate Transition blocks.

Other factors, such as additional blocks in the harness and solver heuristics, can cause test harness execution to differ from the main model. The graphical and compiled interface of the component takes precedence over other execution semantics.

Sample Time Specification

Simulink supports an array of sample times, including types that are derived during model compilation. Test harness construction supports periodic discrete, continuous, and fixed-in-minor-step sample times with these considerations:

- Source blocks that support the desired rate are configured to do so, and the signal conversion subsystem contains a Signal Specification block with the rate specification.
- Test harness construction does not configure source blocks that cannot support the desired rate.

- If the desired rate is periodic discrete or fixed-in-minor-step, the test harness contains a Rate Transition block in the signal conversion subsystem.
- If the desired rate is continuous, the execution semantics are determined by the solver. The signal conversion subsystem does not contain a Rate Transition block.

Other sample time specifications are ignored during test harness construction. In those cases, solver settings determine execution behavior.

See Also

“Create Test Harnesses and Select Properties” on page 2-13

Create Test Harnesses and Select Properties

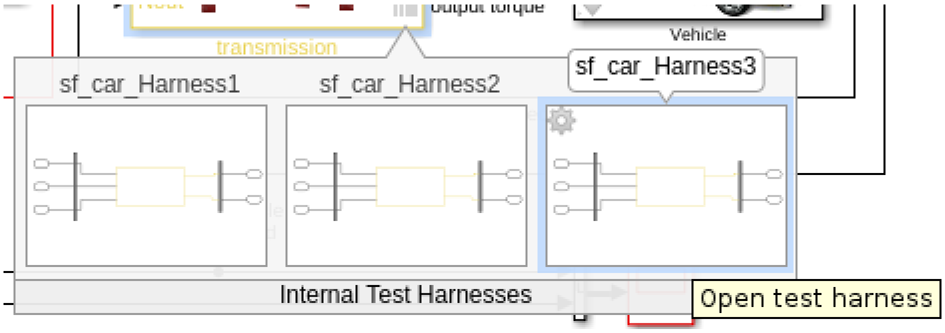
In this section...

Create a Test Harness

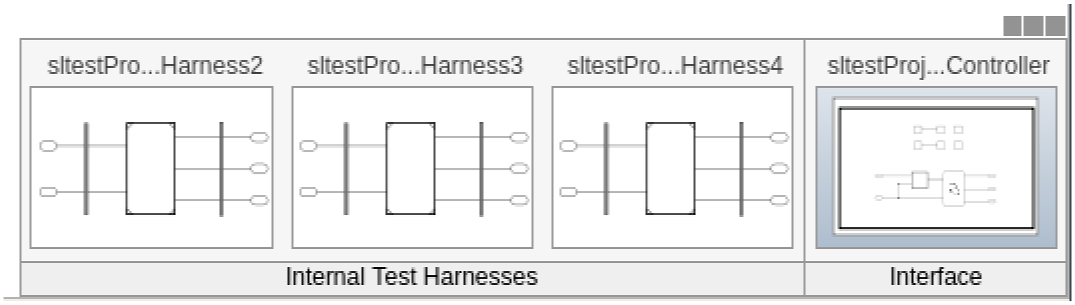
To create a test harness for a top-level model, select **Analysis > Test Harness > Create for Model**. To create a test harness for a subsystem, select the subsystem and select **Analysis > Test Harness > Create for <subsystem name>**. Set test harness properties using the Create Test Harness dialog box.

Preview and Open Test Harnesses

When a model component has a test harness, a badge appears in the lower right of the block. To view the test harnesses, click the badge. To open a test harness, click a tile.



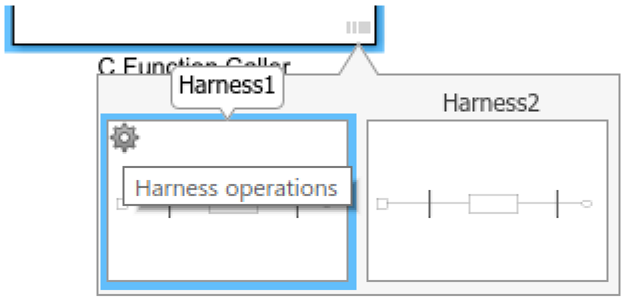
To view test harnesses for a model block diagram, click the pullout icon in the model canvas. To open a test harness, click a tile.



Change Test Harness Properties

To change properties of an open test harness, click the badge in the test harness block diagram and click **Test harness properties** to open the harness properties dialog box.

To change properties of test harnesses from the main model, click the **Harness operations** icon from the test harness preview.



Considerations for Selecting Test Harness Properties

Before selecting test harness properties, consider the following:

- What data source you want to use for your test case input
- How you want to view or store test output
- Whether you want to copy parameters and workspaces from the main model to the harness
- Whether you plan to edit the component under test
- How you want to synchronize changes between the test harness and model

Except for sources and sinks, you can change harness properties later using the harness properties dialog box. To change sources and sinks after harness creation, manually remove the blocks from the test harness and replace them with new sources and sinks.

Harness Name

Test harnesses must use valid MATLAB filenames.

Save Test Harnesses Externally

This option controls how the model stores test harnesses. A model stores all its test harnesses either internally or externally. If a model already has test harnesses, this item states the harness storage type as **Harnesses saved <internally|externally>**.

- When cleared, the model saves test harnesses as part of the model SLX file.

- When selected, the model saves test harnesses in separate SLX files to the current working folder, and adds a harness information XML file to the model folder. The harness information file must remain in the same folder as the model.

See “Manage Test Harnesses” on page 2-28.

Sources and Sinks

In the Create Test Harness dialog box, under **Sources and Sinks**, select the source and sink from the respective menus. The menus provide common sources and sinks.

You can also use source and sink blocks from the Simulink Sources or Sinks library. Select `Custom` source or sink, and enter the path to the block. For example:

```
simulink/Sources/Sine Wave
```

```
simulink/Sinks/Terminator
```

Custom sources and sinks build the test harness with one block per port.

Add scheduler for function-calls and rates / Generate function-call signals using

The title of this option depends on whether the component under test is a subsystem or a model. To include a scheduler block in your test harness, select a block from the drop-down list.

- **Add scheduler for function-calls and rates**: For a model, you can use the block to to call functions and set sample times for model inputs and outputs.
- **Generate function-call signals using**: For a subsystem, you can use the block to call functions in the subsystem.

Enable Initialize, Reset, and Terminate ports

Selecting this option exposes initialize, terminate, or reset function-call ports in the component under test and connects the scheduler block to the ports.

This option appears when you create a test harness for a top-level model and select a block for the **Add scheduler for function-calls and rates** option.

Add Separate Assessment Block

Select **Add separate assessment block** to include a separate Test Assessment block in the test harness.

A Test Assessment block is a separate Test Sequence block configured with properties commonly used for verifying the component under test. For more information, see “Assess Simulation and Compare Output Data” on page 3-10 and “Assess Model Simulation Using verify Statements” on page 3-15.

Open Harness After Creation

Clear **Open Harness After Creation** to create the test harness without opening it. This can be useful creating multiple test harnesses in succession.

Create without compiling the model

Creating a test harness without compiling the model can be useful if you are prototyping a design that cannot yet compile. When you create a test harness without compiling the main model:

- Parameters are not copied to the test harness workspace.
- The main model configuration is not copied to the test harness.
- The test harness does not contain conversion subsystems.

You may need to add blocks such as signal conversion blocks to the test harness. You can rebuild the harness when you are ready to compile the main model. For more information, see “Synchronize Changes Between Test Harness and Model” on page 2-53.

Create scalar inputs

When you select this property, the test harness creates scalar inputs for multidimensional signals. The individual scalar inputs are reshaped to match the dimension of the input signals to the component under test. This option applies to test harnesses with Inport, Constant, Signal Builder, From Workspace, or From File source blocks.

Post-create callback method

You can customize your test harness using a post-create callback. A post-create callback is a function that runs after the harness is created. For example, your callback can set up signal logging, add custom blocks, or change the harness simulation times. For more information, see “Customize Test Harnesses” on page 2-40.

Rebuild harness on open

When you select this property, the test harness rebuilds every time you open it. For details on the rebuild process, see “Synchronize Changes Between Test Harness and Model” on page 2-53.

Update Configuration Parameters and Model Workspace data on rebuild

When you select this property, configuration parameters and model workspace data update when you rebuild the harness. For details on the rebuild process, see “Synchronize Changes Between Test Harness and Model” on page 2-53.

Post-rebuild callback method

You can customize your test harness using a post-rebuild callback. A post-rebuild callback is a function that runs after the harness is rebuilt. For example, your callback can set up signal logging, add custom blocks, or change the harness simulation times. For more information, see “Customize Test Harnesses” on page 2-40.

Synchronization Mode

Synchronization mode controls when changes to the component under test are synced to the main model, and when changes to the harness owner are synced into a test harness.

- On harness open — The component in the test harness is updated when the harness opens. Synchronizing on harness open is useful if you update the design in the main model.
- On harness close — The component in the main model is updated when the harness closes. Synchronizing on harness close is useful if you make design changes in the test harness. Avoid synchronizing on harness close if you want to prevent inadvertent changes to the component in the main model.
- During push — Synchronization occurs manually, by selecting **Analysis > Test Harness > Push Component and Parameters to Main Model**.

- During rebuild — Synchronization occurs manually, by selecting **Analysis** > **Test Harness** > **Rebuild Harness from Main Model**.

Verification Modes

The test harness verification mode determines the type of block generated in the test harness.

- `Normal`: A Simulink block diagram.
- `Software-in-the-Loop (SIL)`: The component under test references generated code, operating as software-in-the-loop. Requires Embedded Coder®.
- `Processor-in-the-Loop (PIL)`: The component under test references generated code for a specific processor instruction set, operating as processor-in-the-loop. Requires Embedded Coder.

Note Keep the SIL or PIL code in the test harness synchronized with the latest component design. If you select SIL or PIL verification mode without selecting **Rebuild harness on open**, your SIL or PIL block code might not reflect recent updates to the main model design. Regenerate code for the SIL or PIL block in the test harness by selecting **Analysis > Test Harness > Rebuild Harness from Main Model**.

See Also

Test Sequence | “Synchronize Changes Between Test Harness and Model” on page 2-53

Refine, Test, and Debug a Subsystem

In this section...

Test harnesses provide a development and testing environment that leaves the main model design intact. You can test a functional unit of your model in isolation without altering the main model. This example demonstrates refining and testing a controller subsystem using a test harness. The main model is a controller-plant model of an air conditioning/heat pump unit. The controller must operate according to several simple requirements.

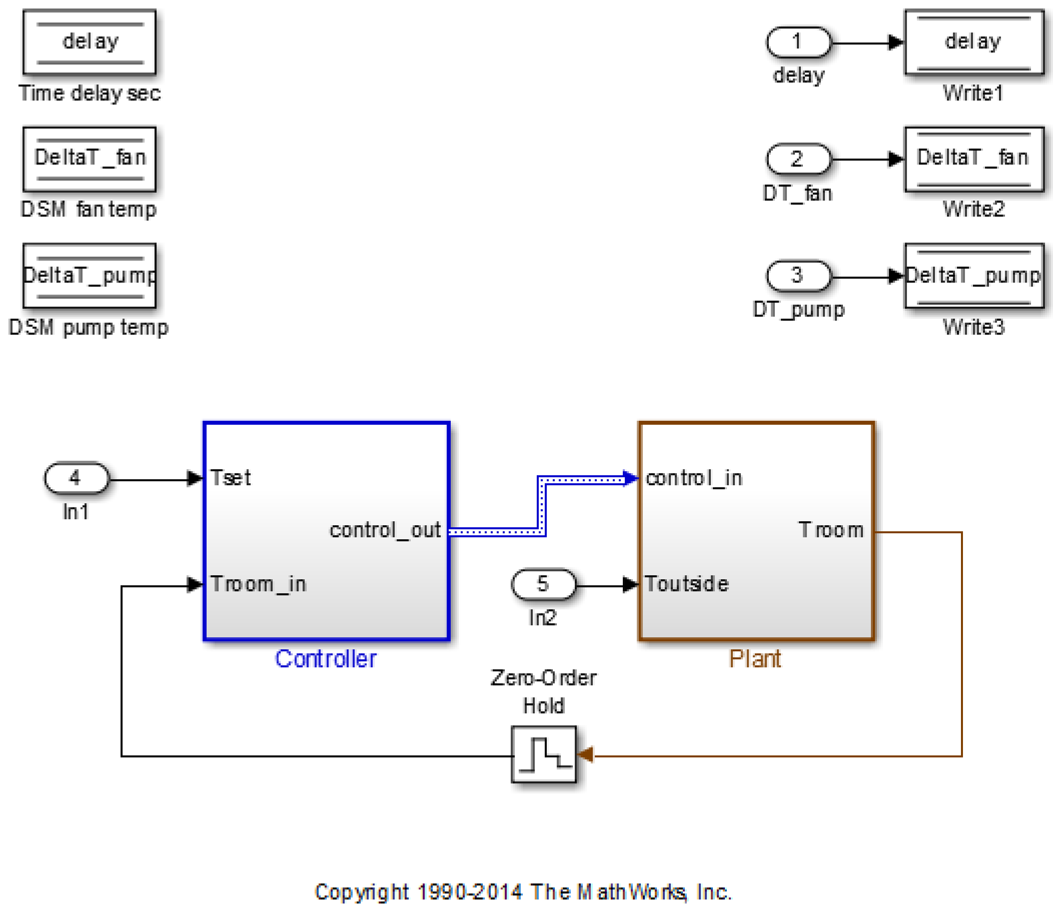
Model and Requirements

1. Access the model. Enter

   ```
   cd(fullfile(docroot,'toolbox','sltest','examples'))
   ```
2. Copy this model file and supporting files to a writable location on the MATLAB path:

   ```
   sltestHeatpumpExample.slx
   sltestHeatpumpBusPostLoadFcn.mat
   PumpDirection.m
   ```
3. Open the model.

   ```
   open_system('sltestHeatpumpExample')
   ```



In the example model:

- The controller accepts the room temperature and the set temperature inputs.
- The controller output is a bus with signals controlling the fan, heat pump, and the direction of the heat pump (heat or cool).
- The plant accepts the control bus. The heat pump and the fan signals are Boolean, and the heat pump direction is specified by +1 for cooling and -1 for heating.

The test covers four temperature conditions. Each condition corresponds to one operating state with fan, pump, and pump direction signal outputs.

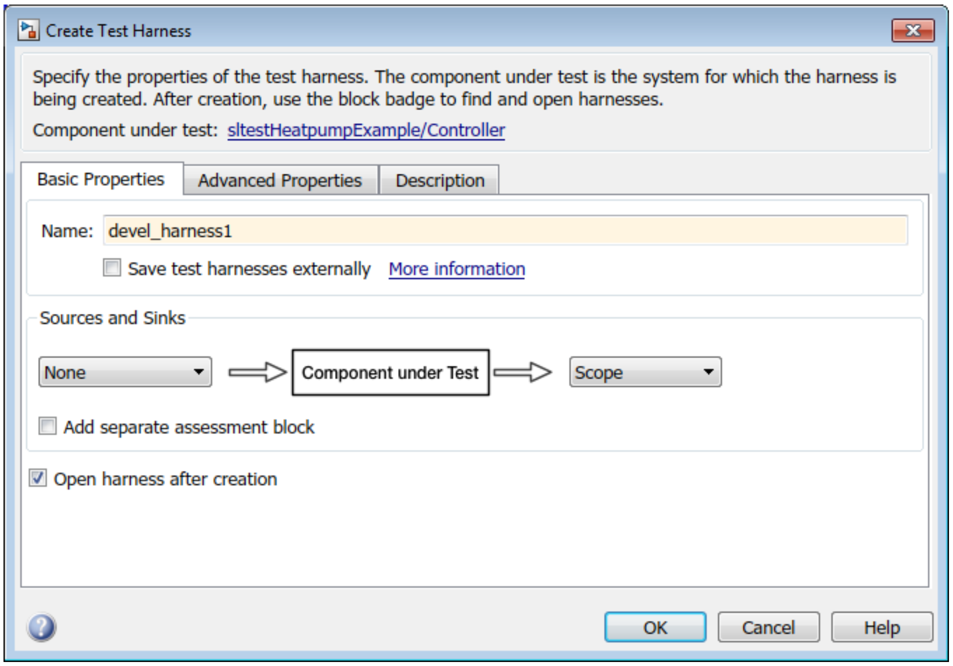
Temperature condition	System state	Fan command	Pump command	Pump direction
`|Troom - Tset| < DeltaT_fan`	idle	0	0	0
`DeltaT_fan <= |Troom - Tset| < DeltaT_pump`	fan only	1	0	0
`|Troom - Tset| >= DeltaT_pump and Tset < Troom`	cooling	1	1	-1
`|Troom - Tset| >= DeltaT_pump and Tset > Troom`	heating	1	1	1

Create a Harness for the Controller

1. Right-click the `Controller` subsystem and select **Test Harness > Create for 'Controller'**.
2. Set the harness properties:

   In the **Basic Properties** tab:

   - **Name**: `devel_harness_1`
   - Clear **Save test harness externally**
   - **Sources and Sinks**: **None** and **Scope**
   - Clear **Add separate assessment block**
   - Select **Open harness after creation**


Create Test Harness

Specify the properties of the test harness. The component under test is the system for which the harness is being created. After creation, use the block badge to find and open harnesses.

Component under test: sltestHeatpumpExample/Controller

Name: devel_harness1

☐ Save test harnesses externally

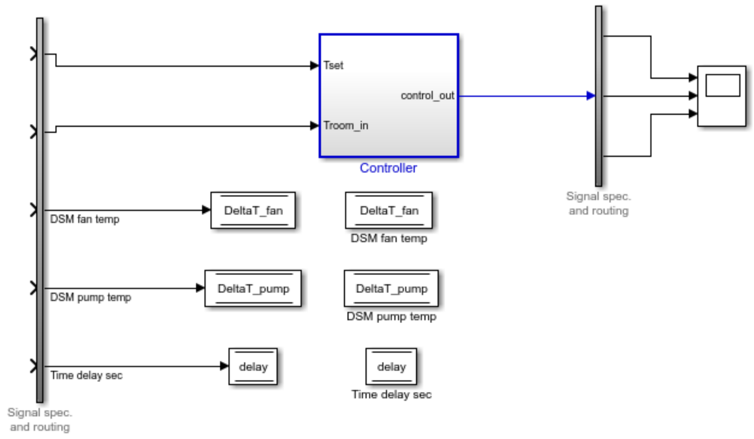
Sources and Sinks

None → Component under Test → Scope

☐ Add separate assessment block

☑ Open harness after creation

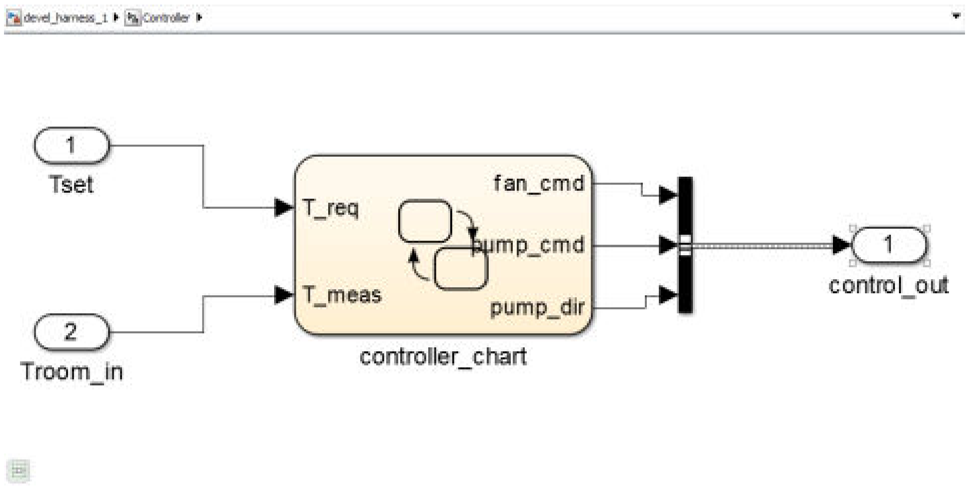
3 Click **OK** to create the test harness.



Inspect and Refine the Controller

1 In the test harness, double-click `Controller` to open the subsystem.

2 Connect the chart to the Inport blocks.



3 In the test harness, click the Save button to save the test harness and model.

Add Test Inputs and Test the Controller

1 Navigate to the top level of `devel_harness_1`.

2 Create a test input for the harness with a constant `Tset` and a time-varying `Troom`. Connect a Constant block to the `Tset` input and set the value to `75`.

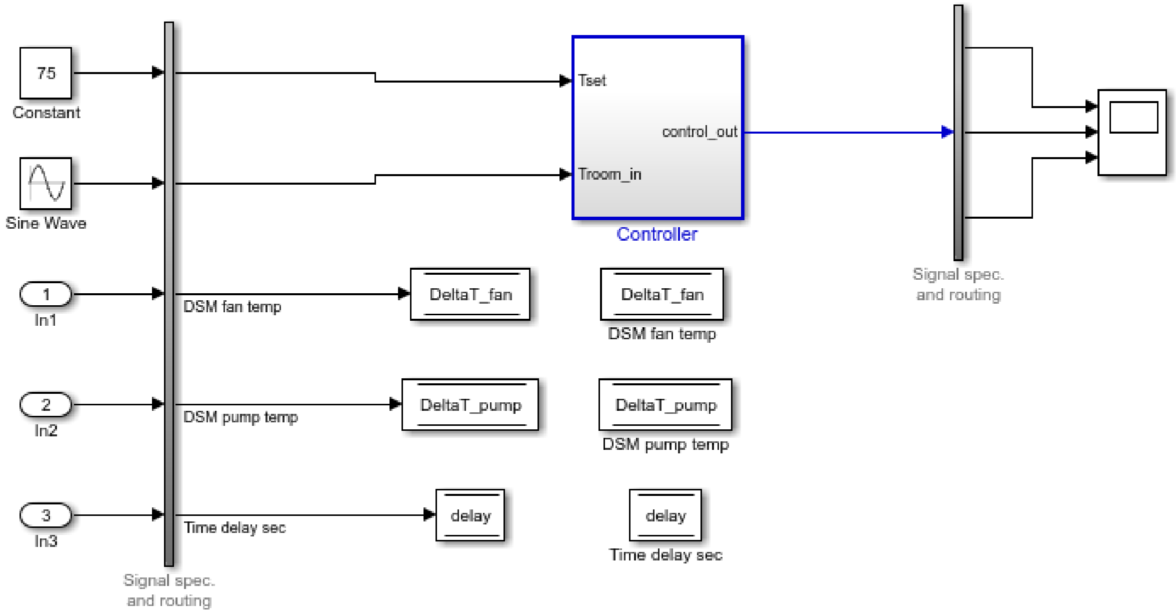
3 Add a Sine Wave block to the harness model to simulate a temperature signal. Connect the Sine Wave block to the conversion subsystem input `Troom_in`.

4 Double-click the Sine Wave block and set the parameters:

- **Amplitude**: `15`
- **Bias**: `75`
- **Frequency**: `2*pi/3600`
- **Phase (rad)**: 0
- **Sample time**: `1`

- Select **Interpret vector parameters as 1-D**.

5 Connect Inport blocks to the Data Store Write inputs.



6 In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select **Input** and enter `u`. `u` is an existing structure in the MATLAB base workspace.

7 In the **Solver** pane, set **Stop time** to `3600`.

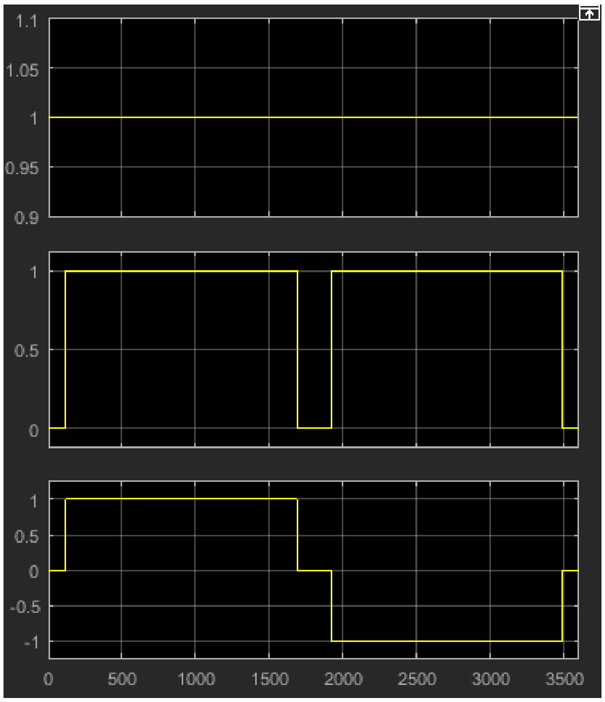
8 Open the scope in the test harness and change the layout to show three plots.

9 Click **Run** to simulate.

Debug the Controller

1 Observe the controller output. `fan_cmd` is `1` during the IDLE condition where `|Troom - Tset| < DeltaT_fan`.

This is a bug. `fan_cmd` should equal `0` at `IDLE`. The `fan_cmd` control output must be changed for `IDLE`.



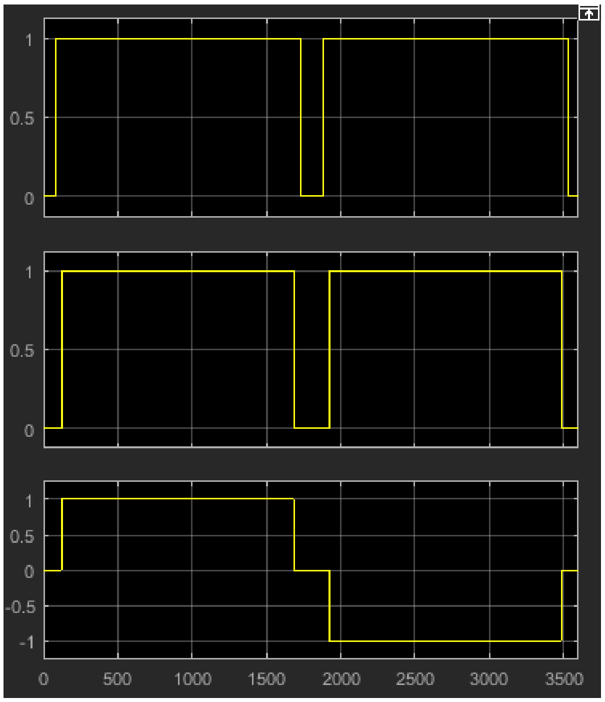
2 In the harness model, open the `Controller` subsystem.

3 Open `controller_chart`.

4 In the `IDLE` state, `fan_cmd` is set to return `1`. Change `fan_cmd` to return `0`. `IDLE` is now:

```
IDLE
entry:
fan_cmd = 0;
 pump_cmd = 0;
 pump_dir = 0;
```

5 Simulate the harness model again and observe the outputs.



6 `fan_cmd` now meets the requirement to equal `0` at `IDLE`.

See Also

Related Examples

- "Test Downshift Points of a Transmission Controller" on page 3-72

Manage Test Harnesses

In this section...

Internal and External Test Harnesses

You can save test harnesses internally as part of your model SLX file, or externally in separate SLX files. A model stores all test harnesses either internally or externally; it is not possible to use both types of harness storage in one model. You select internal or external test harness storage when you create the first test harness. If your model already has test harnesses, you can convert between the harness storage types.

If you store your model in a change control system, consider using external test harnesses. External test harnesses enable you to create or change a harness without changing the model file. If you plan to share your model often, consider using internal test harnesses to simplify file management. Creating or changing an internal test harness changes your model SLX file. Both internal and external test harnesses offer the same synchronization, push, rebuild, and badge interface functionality.

See “Create Test Harnesses and Select Properties” on page 2-13.

Manage External Test Harnesses

Harnesses stored externally use a separate SLX file for each harness, and a `<modelName>_harnessInfo.xml` file containing metadata linking the model and the harnesses. Changing test harnesses can change the `harnessInfo.xml` file.

Follow these guidelines for external test harnesses:

Warning Do not delete the `harnessInfo.xml` file. Deleting the `harnessInfo.xml` file terminates the relationship between the model and harnesses, which cannot be regenerated from the model.

- The `harnessInfo.xml` file must be writable to save changes to the test harness or the main model.
- Keep the `harnessInfo.xml` file in the same folder as the main model. If the `harnessInfo.xml` file and the model are in separate folders, the main model opens but does not present the test harnesses.
- Directories containing test harness SLX files must be on the MATLAB path.
- If you convert internal test harnesses to external test harnesses, the new SLX files save to the current working folder.
- If you convert external test harnesses to internal test harnesses, the external SLX files can be anywhere on the MATLAB path.
- If your model uses external test harnesses, only create a copy of your model using **File > Save As** from the model menu. Using **File > Save As** copies external test harnesses to the destination folder of the new model and keeps the harness information current.

  Copying the model file on disk will not copy external harnesses associated with the model.
- Only change or delete test harnesses using the Simulink UI or commands:

  - To delete test harnesses, use the thumbnail UI or the `sltest.harness.delete` command.
  - To rename test harnesses, use the harness properties UI or the `sltest.harness.set` command.
  - To make a copy of an externally saved test harness, use the `sltest.harness.clone` command or save the test harness to a new name using **File > Save As**.

  Deleting or renaming harness files outside of Simulink causes an inaccurate `harnessInfo.xml` file and problems loading test harnesses.

Convert Between Internal and External Test Harnesses

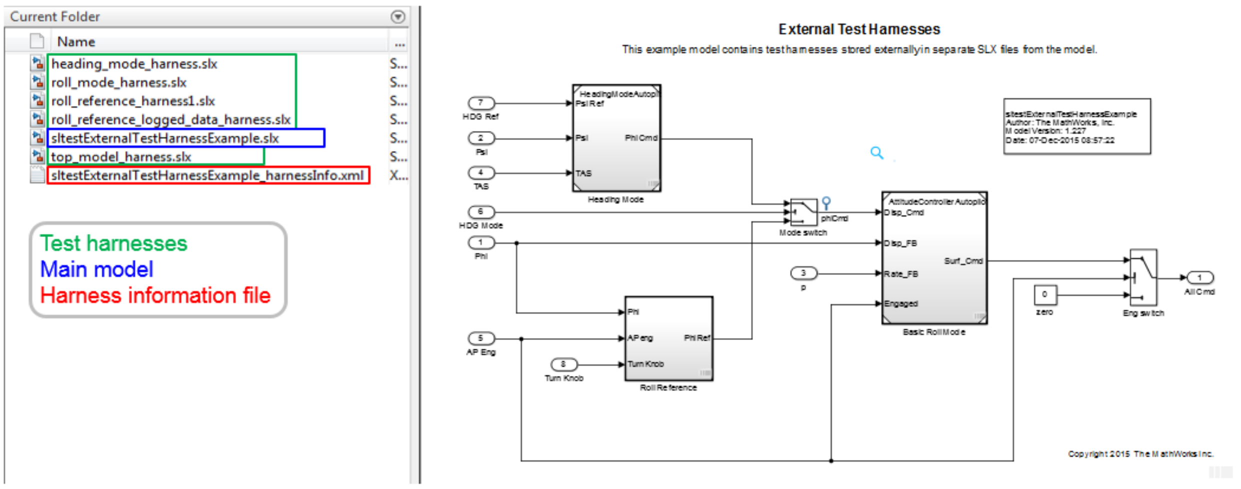
You can change how your model stores test harnesses at different phases of your model lifecycle. For example:

- Develop your model using internal test harnesses so that you can more easily share the model for review. When you complete your design and place the model under change control, convert to external harnesses.
- Use the change-controlled model as the starting point for a new design. Test the existing model with external harnesses to avoid modifying it. Then, create a copy of the existing model. Convert to internal harnesses for the new development phase.

To change the test harness storage to external (or internal):

1. Navigate to the top of the main model.
2. Select **Analysis** > **Test Harness** > **Convert To External (Internal) Harnesses**.
3. A dialog box provides information on the conversion procedure and the affected test harnesses. Click **Yes** to continue.

   The harnesses are converted.
4. The conversion to external test harnesses creates an SLX file for each test harness and a harness information XML file `<modelName>_harnessInfo.xml`.



Current Folder

Name	...
heading_mode_harness.slx	S...
roll_mode_harness.slx	S...
roll_reference_harness1.slx	S...
roll_reference_logged_data_harness.slx	S...
sltestExternalTestHarnessExample.slx	S...
top_model_harness.slx	S...
sltestExternalTestHarnessExample_harnessInfo.xml	X...

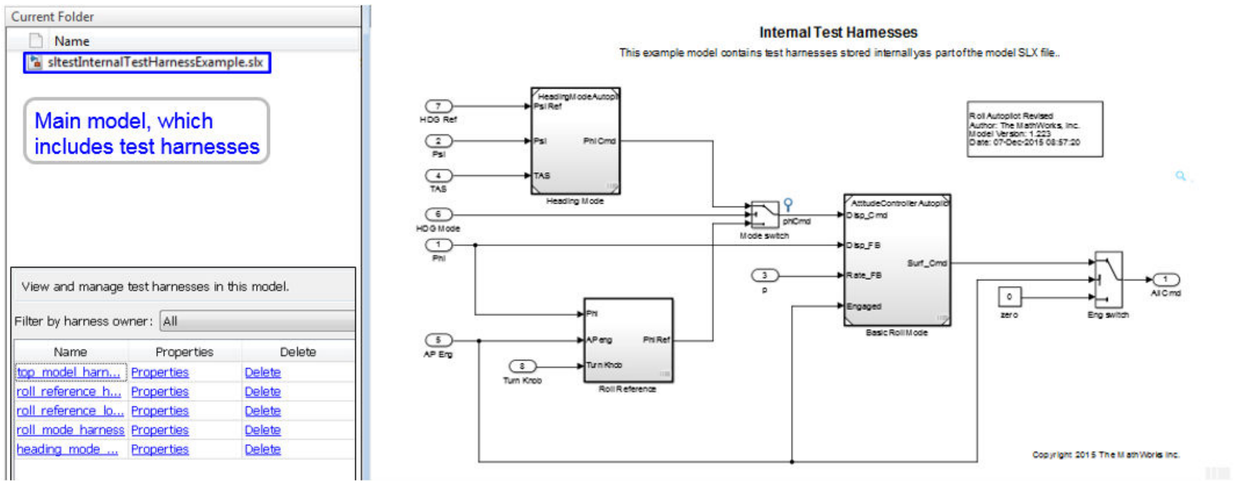
Test harnesses
Main model
Harness information file

External Test Harnesses

This example model contains test harnesses stored externally in separate SLX files from the model.

sltestExternalTestHarnessExample
Author: The MathWorks, Inc.
Model Version: 1.227
Date: 07-Dec-2015 08:57:22

Inversely, conversion to internal test harnesses moves the test harness SLX files and the `harnessInfo.xml` file.


Current Folder
Name
sltestInternalTestHarnessExample.slx
Main model, which includes test harnesses
Internal Test Harnesses
This example model contains test harnesses stored internally as part of the model SLX file.
Heading Mode
Mode switch
Basic Roll Mode
Roll Reference
Eng switch
Copyright 2015 The MathWorks Inc.

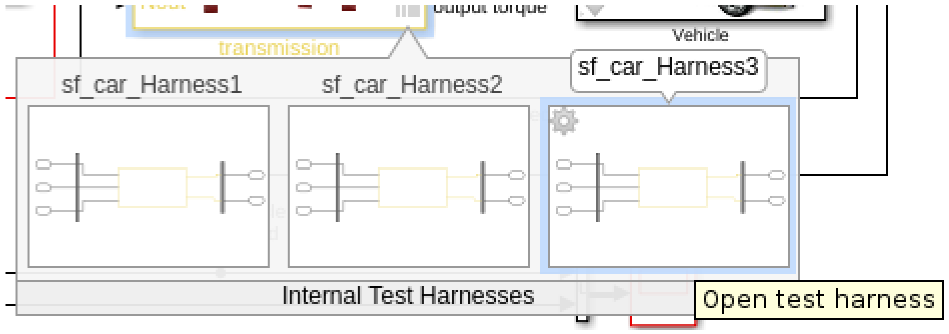
View and manage test harnesses in this model.

Filter by harness owner: All

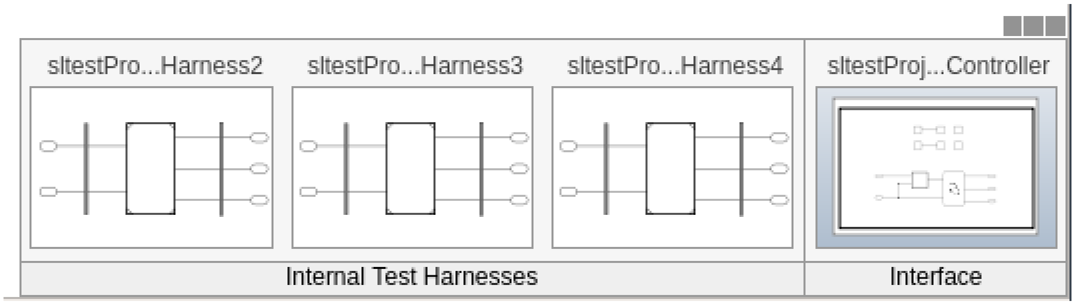
Name	Properties	Delete
top_model_harn...	Properties	Delete
roll_reference_h...	Properties	Delete
roll_reference_lo...	Properties	Delete
roll_mode_harness	Properties	Delete
heading_mode_...	Properties	Delete

Preview and Open Test Harnesses

When a model component has a test harness, a badge appears in the lower right of the block. To view the test harnesses, click the badge. To open a test harness, click a tile.

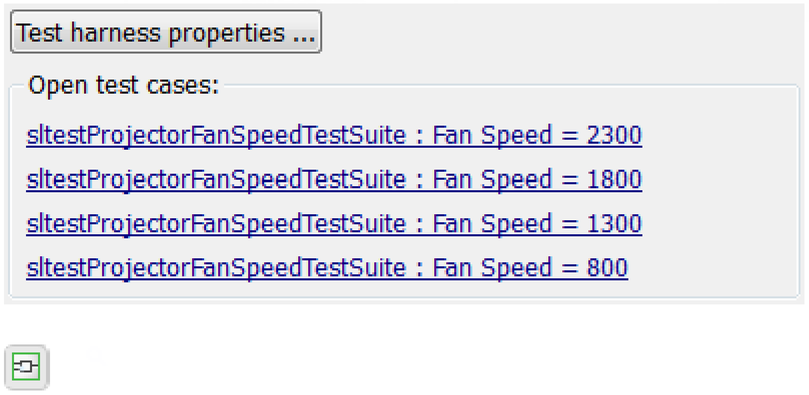


To view test harnesses for a model block diagram, click the pullout icon in the model canvas. To open a test harness, click a tile.



Find Test Cases Associated with a Test Harness

To list open test cases that refer to the test harness, click the badge in the test harness canvas. You can click a test case name and navigate to the test case in the Test Manager.



Open test cases:

sltestProjectorFanSpeedTestSuite : Fan Speed = 2300

sltestProjectorFanSpeedTestSuite : Fan Speed = 1800

sltestProjectorFanSpeedTestSuite : Fan Speed = 1300

sltestProjectorFanSpeedTestSuite : Fan Speed = 800

Export Test Harnesses to Separate Models

You can export test harnesses to separate models, which is useful for archiving test harnesses or sharing a test harness design without sharing the model.

- To export an individual test harness:

1. From the test harness menu, select **Analysis** > **Test Harness** > **Detach and Export Harness**.
2. A dialog box confirms the harness export. Click **OK**.
3. Enter a file name for the separate model.

   The harness converts to a separate model. Converting removes the harness from the main model and breaks the relationship to the main model.

- To export all harnesses in a model:

  1. Navigate to the top level of the test harness.
  2. Select no blocks.
  3. From the model menu, select **Analysis** > **Test Harness** > **Detach and Export Harnesses**.
  4. A dialog box confirms the harness export. Click **OK**.

     The harnesses convert to separate models. Converting removes the harnesses from the main model and breaks the relationships to the main model.

See `sltest.harness.export`.

Clone and Export a Test Harness to a Separate Model

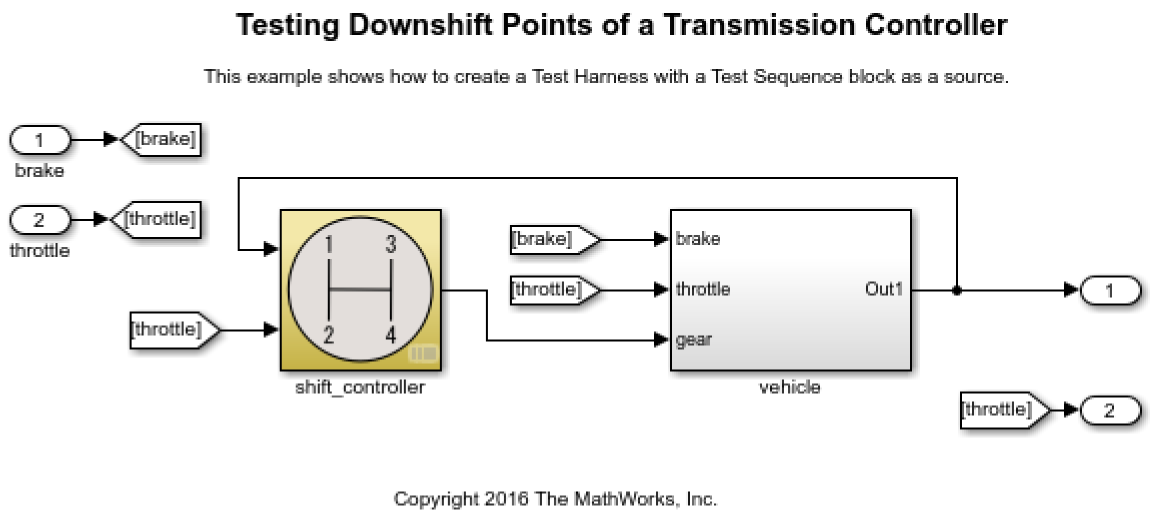
This example demonstrates cloning an existing test harness and exporting the cloned harness to a separate model. This can be useful if you want to create a copy of a test harness as a separate model, but leave the test harness associated with the model component.

High-level Workflow

1. If you don't know the exact properties of the test harness you want to clone, get them using sltest.harness.find. You need the harness owner ID and the harness name.
2. Clone the test harness using sltest.harness.clone.
3. Export the test harness to a separate model using sltest.harness.export. Note that there is no association between the exported model and the original model. The exported model stands alone.

Open the Model and Save a Local Copy

```
model = 'sltestTestSequenceExample';
open_system(model)
```


Testing Downshift Points of a Transmission Controller
This example shows how to create a Test Harness with a Test Sequence block as a source.
1
brake
[brake]
2
throttle
[throttle]
[throttle]
1
3
2
4
shift_controller
[brake]
[throttle]
brake
throttle
gear
Out1
vehicle
1
[throttle]
2
Copyright 2016 The MathWorks, Inc.

Save the local copy in a writable location on the MATLAB path.

Get the Properties of the Source Test Harness

```
properties = sltest.harness.find([model '/shift_controller'])
```

```
properties = 

  struct with fields:

                model: 'sltestTestSequenceExample'
                 name: 'controller_harness'
          description: ''
                 type: 'Testing'
          ownerHandle: 10.0167
        ownerFullPath: 'sltestTestSequenceExample/shift_controller'
            ownerType: 'Simulink.SubSystem'
               isOpen: 0
          canBeOpened: 1
```

```
              lockMode: 0
      verificationMode: 0
        saveExternally: 0
         rebuildOnOpen: 0
      rebuildModelData: 0
   postRebuildCallback: ''
             graphical: 0
               origSrc: 'Test Sequence'
              origSink: 'Test Assessment'
   synchronizationMode: 0
```

Clone the Test Harness

Clone the test harness using sltest.harness.clone, the `ownerFullPath` and the `name` fields of the harness properties structure.

```
sltest.harness.clone(properties.ownerFullPath,properties.name,'ControllerHarness2')
```

Save the Model

Before exporting the harness, save changes to the model.

```
save_system(model)
```

Export the Test Harness to a Separate Model

Export the test harness using sltest.harness.export. The exported model name is `ControllerTestModel`.

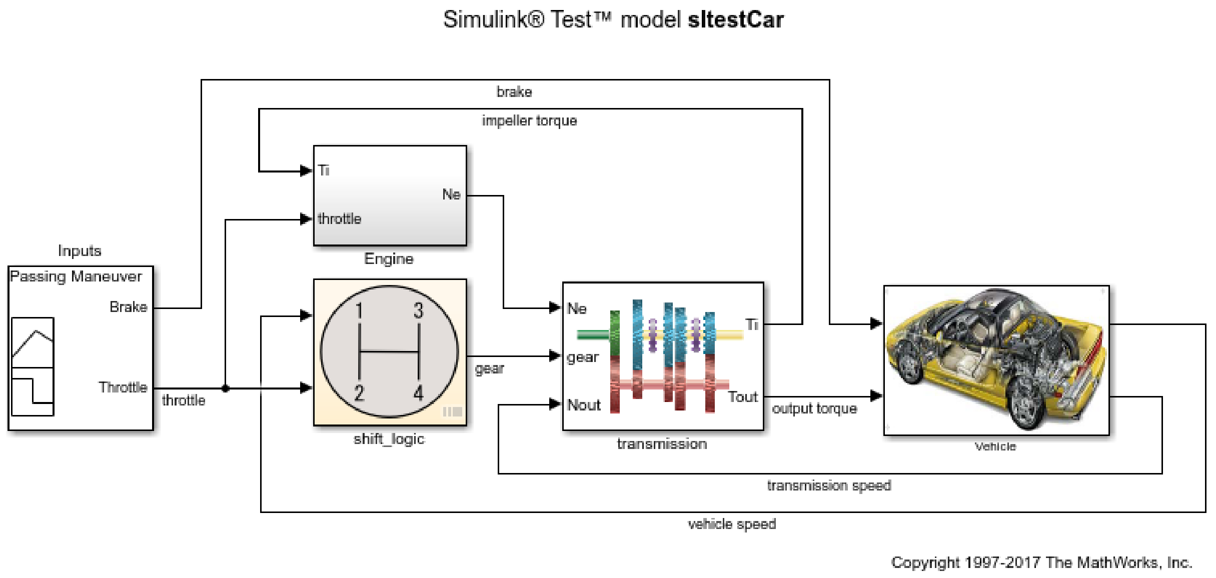
```
sltest.harness.export([model '/shift_controller'],'ControllerHarness2',...
    'Name','ControllerTestModel')

clear('model')
clear('properties')
close_system('sltestTestSequenceExample',0)
```

Delete Test Harnesses Programmatically

This example shows how to delete test harnesses programmatically. Deleting with % the programmatic interface can be useful when your model has multiple test harnesses at different hierarchy levels. This example demonstrates creating four test harnesses, then deleting them.

1. Open the model

```
open_system('sltestCar');
```



2. Create two harnesses for the `transmission` subsystem, and two harnesses for the `transmission ratio` subsystem.

```
sltest.harness.create('sltestCar/transmission');
sltest.harness.create('sltestCar/transmission');
sltest.harness.create('sltestCar/transmission/transmission ratio');
sltest.harness.create('sltestCar/transmission/transmission ratio');
```

3. Find the harnesses in the model.

```
test_harness_list = sltest.harness.find('sltestCar')
```

```
test_harness_list = 

  1x5 struct array with fields:

    model
    name
    description
    type
    ownerHandle
    ownerFullPath
    ownerType
    isOpen
    canBeOpened
    lockMode
    verificationMode
    saveExternally
    rebuildOnOpen
    rebuildModelData
    postRebuildCallback
    graphical
    origSrc
    origSink
    synchronizationMode
```

4. Delete the harnesses.

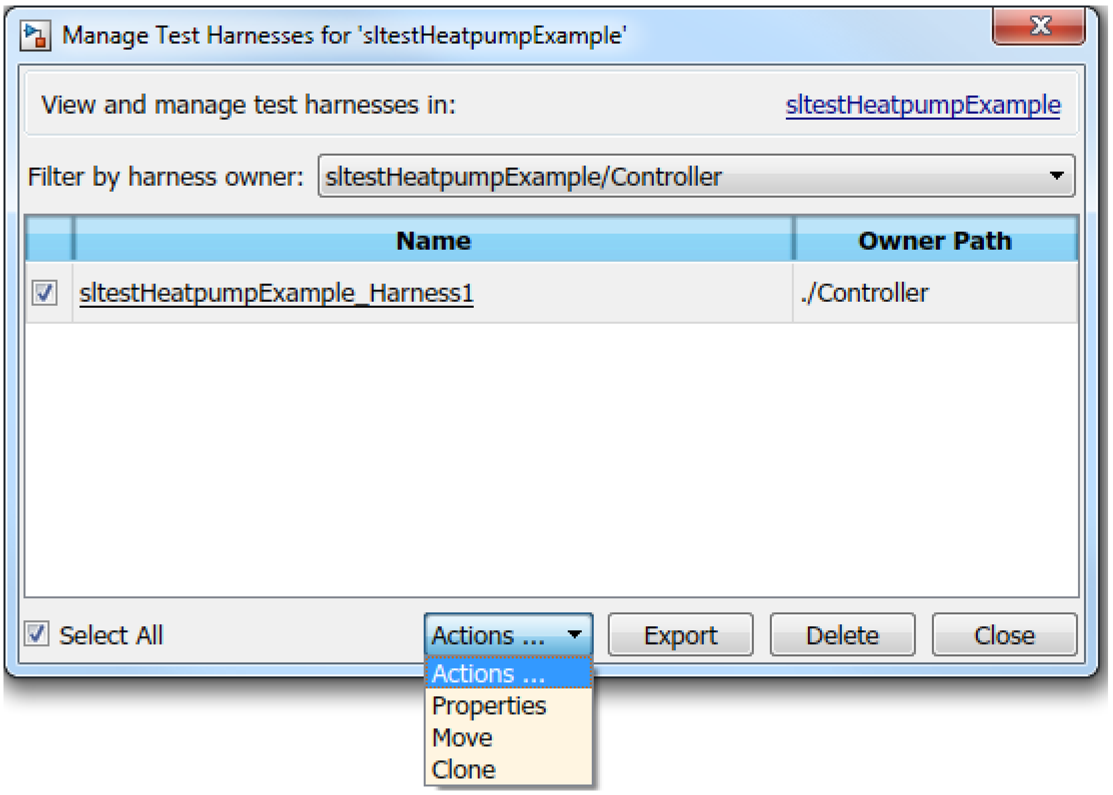
```
for k = 1:length(test_harness_list)
     sltest.harness.delete(test_harness_list(k).ownerFullPath,...
     test_harness_list(k).name)
end

close_system('sltestCar',0);
```

Move and Clone Test Harnesses

Simulink Test gives you the ability to move/clone test harnesses from a source owner to a destination owner without having to compile the model. You can move or clone:

- Subsystem harnesses across subsystems. The destination subsystem could also be in a different model.
- Harnesses for library components across libraries.

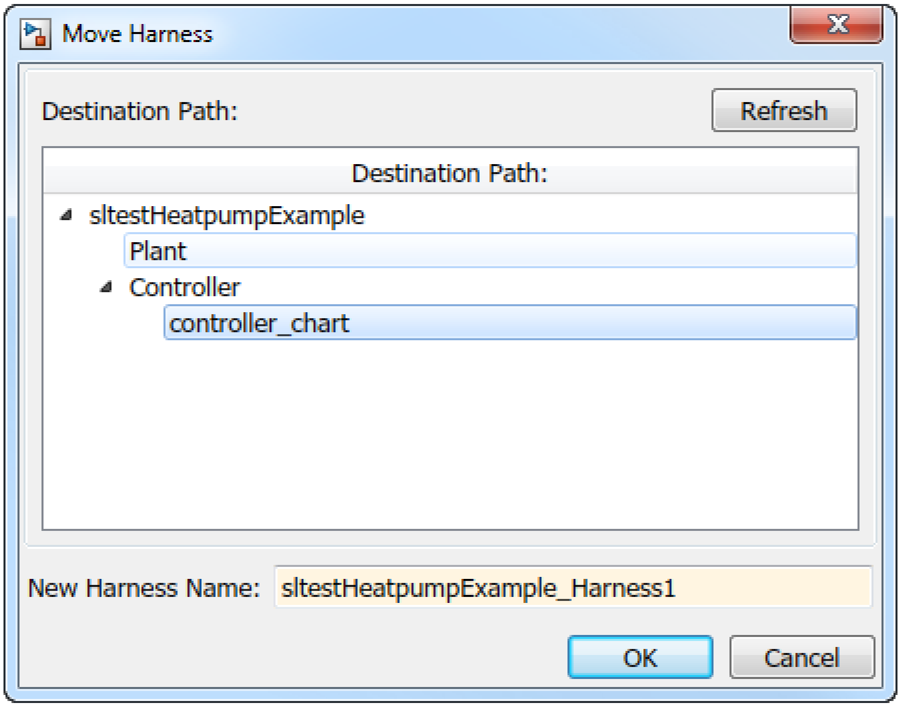
To move or clone harnesses, right-click the Simulink canvas and select **Test Harness > Manage Test Harnesses**. The Manage Test Harness dialog box opens and lists the test harnesses associated with the subsystem/block specified in **Filter by harness owner**. Click **Actions** to access the Move and Clone options.


Manage Test Harnesses for 'sltestHeatpumpExample'

View and manage test harnesses in: sltestHeatpumpExample

Filter by harness owner: sltestHeatpumpExample/Controller

	Name	Owner Path
✓	sltestHeatpumpExample_Harness1	./Controller



New Harness Name: sltestHeatpumpExample_Harness1

Select the destination path and name your test harness.

See Also

Functions

`sltest.harness.clone` | `sltest.harness.create` | `sltest.harness.delete` | `sltest.harness.export` | `sltest.harness.find` | `sltest.harness.load` | `sltest.harness.move` | `sltest.harness.open`

Customize Test Harnesses

In this section...

You can customize a test harness by using a function that runs after creating or rebuilding the test harness. In the function, script the commands to customize your test harness. For example, the function can:

- Connect custom source or sink blocks.
- Add a plant subsystem for closed-loop testing.
- Change the configuration set.
- Enable signal logging.
- Change the simulation stop time.

The test harness customization function runs as a test harness post-create callback or post-rebuild callback. To customize a test harness using a callback function:

1. Create the callback function.
2. In the function, use the Simulink programmatic interface to script the commands to customize the test harness. For more information, see the functions listed in “Programmatic Model Editing” (Simulink).
3. Specify the function as the post-create or post-rebuild callback:

   - For a new test harness,

     - If you are using the UI, enter the function name in the **Post-create callback method** or **Post-rebuild callback method** in the **Advanced Properties** of the harness creation dialog box.
     - If you are using `sltest.harness.create`, specify the function as the `PostCreateCallback` or `PostRebuildCallback` value.
   - For an existing test harness,

     - If you are using the UI, enter the function name in **Post-rebuild callback method** in the harness properties dialog box.

- If you are using `sltest.harness.set`, specify the function as the `PostRebuildCallback` value.

For more information on test harness properties, see “Create Test Harnesses and Select Properties” on page 2-13.

Callback Function Definition and Harness Information

The callback function declaration is

```
function myfun(x)
```

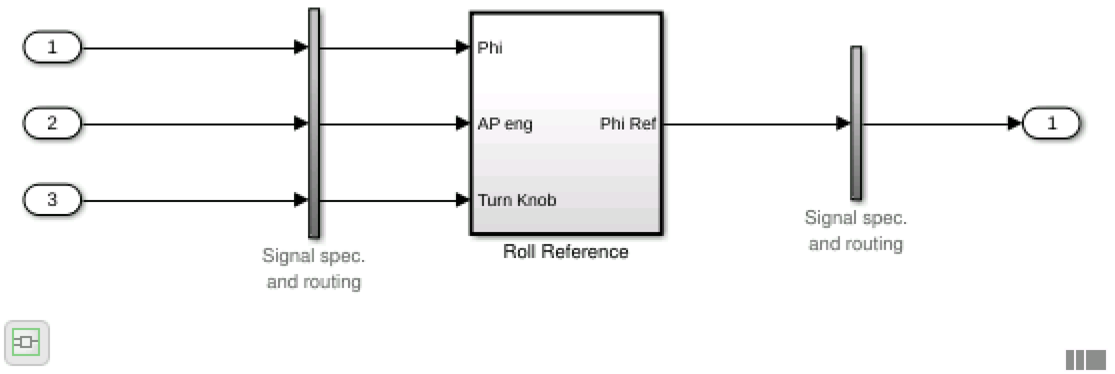
where `myfun` is the function name and `myfun` accepts input `x`. `x` is a struct of information about the test harness automatically created when the test harness uses the callback. You can choose the function and argument names.

For example, define a harness callback function `harnessCustomization.m`:

```
function harnessCustomization(harnessInfo)

% Script commands here to customize your test harness.

end
```

In this example, `harnessInfo` is the struct name and `harnessCustomization` is the function name. When the create or rebuild operation calls `harnessCustomization`, `harnessInfo` is populated with information about the test harness, including handles to the test harness model, main model, and blocks in the test harness.

For example, using `harnessCustomization` as a callback for the following test harness:



populates `harnessInfo` with handles to three sources, one sink, the main model, harness model, harness owner, component under test, and conversion subsystems:

```
harnessInfo = 

  struct with fields:

                    MainModel: 2.0001
                 HarnessModel: 1.1290e+03
                        Owner: 17.0001
                   HarnessCUT: 201.0110
              DataStoreMemory: []
                DataStoreRead: []
               DataStoreWrite: []
                         Goto: []
                         From: []
                      GotoTag: []
       SimulinkFunctionCaller: []
         SimulinkFunctionStub: []
                      Sources: [1.1530e+03 1.1540e+03 1.1550e+03]
                        Sinks: 1.1630e+03
              AssessmentBlock: []
     InputConversionSubsystem: 1.1360e+03
    OutputConversionSubsystem: 1.1560e+03
                   CanvasArea: [215 140 770 260]
```

Use the struct fields to customize the test harness. For example:

- To add a Constant block named `ConstInput` to the test harness, get the name of the test harness model, then use the `add_block` function.

  ```
  harnessName = get_param(harnessInfo.HarnessModel,'Name');
  block = add_block('simulink/Sources/Constant',[harnessName '/ConstInput']);
  ```
- To get the port handles for the component under test, get the `'PortHandles'` parameter for `harnessInfo.HarnessCUT`.

  ```
  CUTPorts = get_param(harnessInfo.HarnessCUT,'PortHandles');
  ```
- To get the simulation stop time for the test harness, get the `'StopTime'` parameter for `harnessInfo.HarnessModel`.

  ```
  st = get_param(harnessInfo.HarnessModel,'StopTime');
  ```
- To set a `16` second simulation stop time for the test harness, set the `'StopTime'` parameter for `harnessInfo.HarnessModel`.

```
set_param(harnessInfo.HarnessModel,'StopTime','16');
```

How to Display Harness Information struct Contents

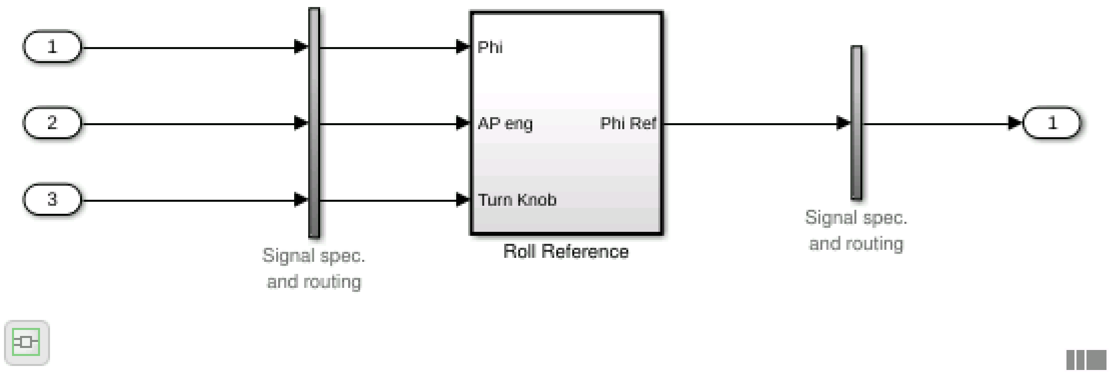
To list the harness information for your test harness:

1. In the callback function, add the line

   ```
   disp(harnessInfo)
   ```
2. Create or rebuild a test harness using the callback function.
3. When you create or rebuild the test harness, the harness information structure contents are displayed on the command line.

Customize a Test Harness to Create Mixed Source Types

This example harness callback function connects a Constant block to the third component input of this example test harness.

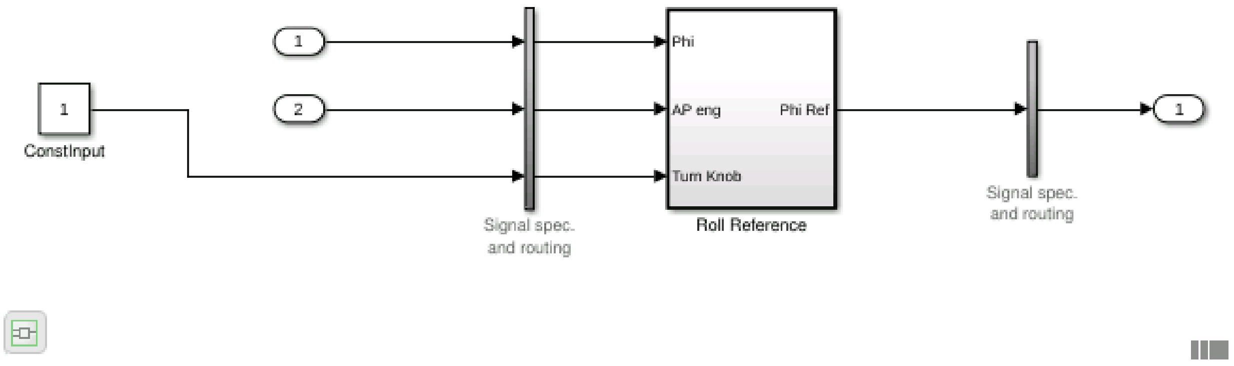


The function follows the procedure:

1. Get the harness model name.
2. Add a Constant block.
3. Get the port handles for the Constant block.
4. Get the port handles for the input conversion subsystem.
5. Get the handles for lines connected to the input conversion subsystem.
6. Delete the existing Inport block.
7. Delete the remaining line.

8 Connect a new line from the Constant block to input 3 of the input conversion subsystem.

```
function harnessCustomization(harnessInfo)

% Get harness model name:
harnessName = get_param(harnessInfo.HarnessModel,'Name');

% Add Constant block:
constBlock = add_block('simulink/Sources/Constant',[harnessName '/ConstInput']);

% Get handles for relevant ports and lines:
constPorts = get_param(constBlock,'PortHandles');
icsPorts = get_param(harnessInfo.InputConversionSubsystem,'PortHandles');
icsLineHandles = get_param(harnessInfo.InputConversionSubsystem,'LineHandles');

% Delete the existing Inport block and the adjacent line:
delete_block(harnessInfo.Sources(3));
delete_line(icsLineHandles.Inport(3));

% Connect the Constant block to the input conversion subsystem:
add_line(harnessInfo.HarnessModel,constPorts.Outport,icsPorts.Inport(3),...
'autorouting','on');

end
```



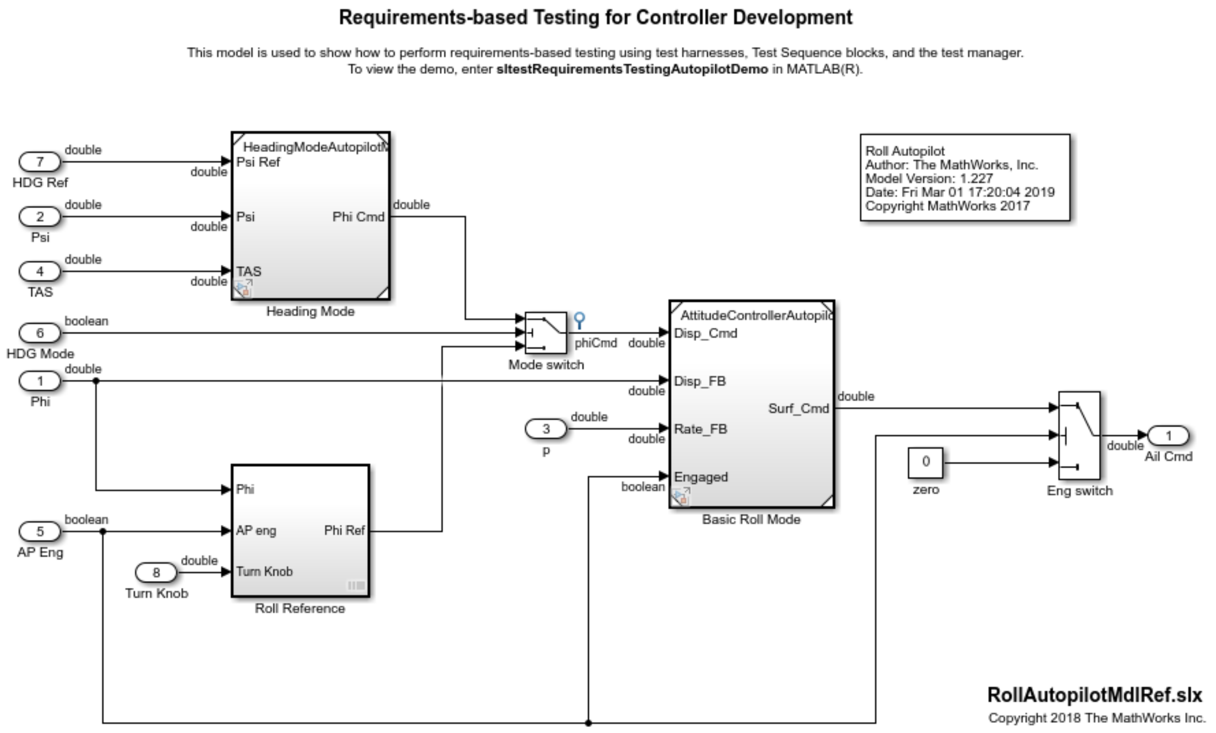
Test Harness Callback Example

This example shows how to use a post-create callback to customize a test harness. The callback changes one harness source from an Inport block to Constant block and enables signal logging in the test harness.

The Model

In this example, you create a test harness for the `Roll Reference` subsystem.

```
open_system('RollAutopilotMdlRef')
```


Requirements-based Testing for Controller Development
This model is used to show how to perform requirements-based testing using test harnesses, Test Sequence blocks, and the test manager.
To view the demo, enter sltestRequirementsTestingAutopilotDemo in MATLAB(R).
HDG Ref
Psi
TAS
HDG Mode
Phi
AP Eng
Turn Knob
Heading Mode
Mode switch
Roll Reference
Basic Roll Mode
Eng switch
zero
Ail Cmd
Roll Autopilot
Author: The MathWorks, Inc.
Model Version: 1.227
Date: Fri Mar 01 17:20:04 2019
Copyright MathWorks 2017
RollAutopilotMdlRef.slx
Copyright 2018 The MathWorks Inc.

Get Path to the Harness Customization Function

```
cbFile = fullfile(matlabroot,'examples','simulinktest','main',...
    'harnessSourceLogCustomization.m');
```

The Customization Function and Test Harness Information

The function `harnessSourceLogCustomization` changes the third source block, and enables signal logging on the component under test inputs and outputs. You can read the function by entering:

```
type(cbFile)
```

The function uses an argument. The argument is a struct listing test harness information. The information includes handles to blocks in the test harness, including:

- Component under test
- Input subsystems
- Sources and sinks
- The harness owner in the main model

For example, `harnessInfo.Sources` lists the handles to the test harness source blocks.

Create the Customized Test Harness

1. Copy the harness customization function to the temporary working directory.

```
copyfile(cbFile,tempdir);
cd(tempdir);
```

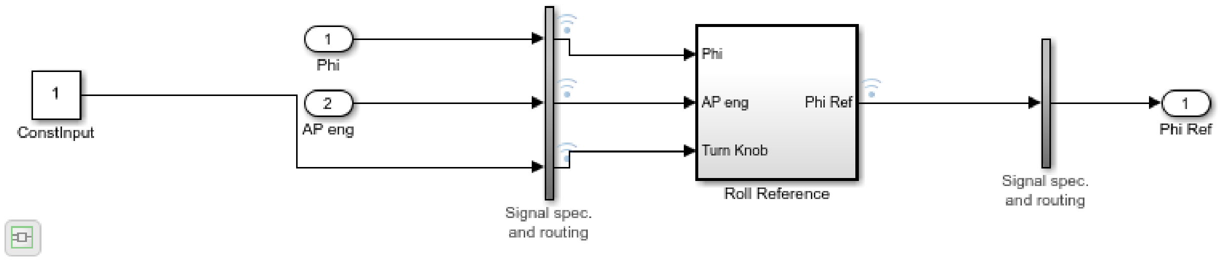
2. In the `RollAutopilotMdlRef` model, right-click the `Roll Reference` subsystem and select **Test Harness > Create for Roll Reference**.

3. In the harness creation dialog box, for **Post-create callback method**, enter `harnessSourceLogCustomization`.

4. Click OK to create the test harness. The harness shows the signal logging and simulation stop time specified in the callback function.

You can also use the `sltest.harness.create` function to create the test harness, specifying the callback function with the `'PostCreateCallback'` name-value pair.

```
sltest.harness.create('RollAutopilotMdlRef/Roll Reference',...
    'Name','LoggingHarness',...
    'PostCreateCallback','harnessSourceLogCustomization');

sltest.harness.open('RollAutopilotMdlRef/Roll Reference','LoggingHarness');
```



```
close_system('RollAutopilotMdlRef',0);
```

See Also

`sltest.harness.create` | `sltest.harness.set`

Related Examples

- “Create Test Harnesses and Select Properties” on page 2-13

Create Test Harnesses from Standalone Models

In this section...

Standalone test models are often used to verify your main model. You can create Simulink Test test harnesses by importing your standalone test models. Importing standalone models enables synchronization and management features, allowing you to:

- Iterate on your design, using model and test harness synchronization
- Manage test harnesses, using the UI and programmatic interface
- Clarify ownership of a test harness by a model, subsystem, or library being tested

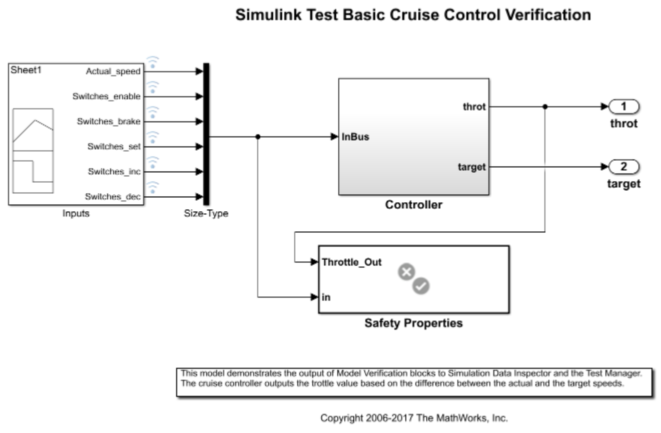
A common test model passes input signals to a copy of a subsystem or a Model block referencing your main model. Test models include models created by Simulink Coverage™ and Simulink Design Verifier™.

Test Harness Import Workflow

Before importing a standalone model as a test harness, determine:

- In the main model, the model or component to associate the test harness with.
- The path to the standalone model.
- The tested component in the standalone model.

  For example, this standalone model tests the `Controller` subsystem. The model passes `Inputs` to `Controller`. `Safety Properties` verifies the `Controller` output.



This model demonstrates the output of Model Verification blocks to Simulation Data Inspector and the Test Manager.
The cruise controller outputs the trottle value based on the difference between the actual and the target speeds.

Copyright 2006-2017 The MathWorks, Inc.

Component Compatibility for Test Harness Import

When you import a model as a test harness, the component in the main model must be compatible with the component in the standalone model.

Component Compatibility for Test Harness Import

In the main model, if the component is:	In the standalone model, the tested component must be:
A user-defined function block (e.g. an S-Function block)	The same block type
The top-level model	A Model block or a subsystem
A subsystem	A subsystem, Model block, or a user-defined function block
A Model block	A Model block or a subsystem

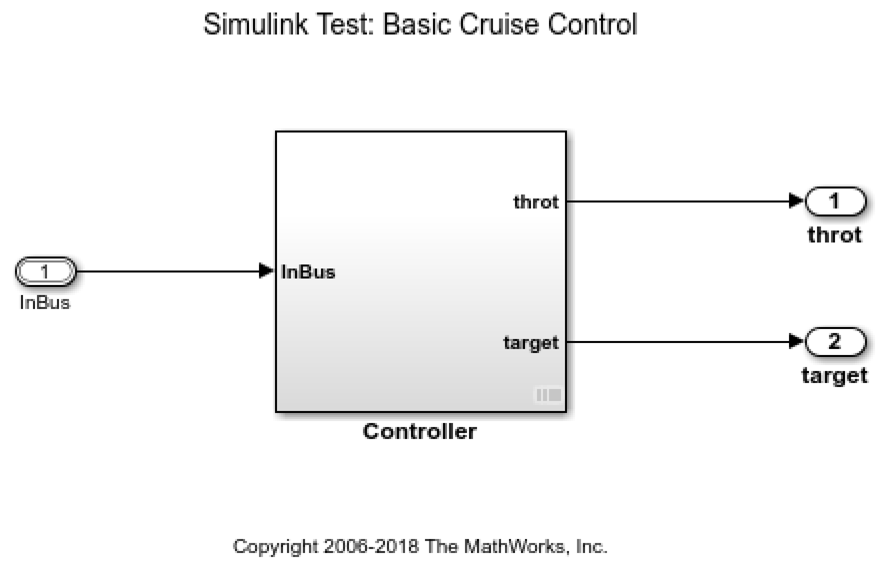
You cannot create a test harness by importing:

- Libraries
- Models that have existing test harnesses
- Models with unsaved changes. Save open models before importing

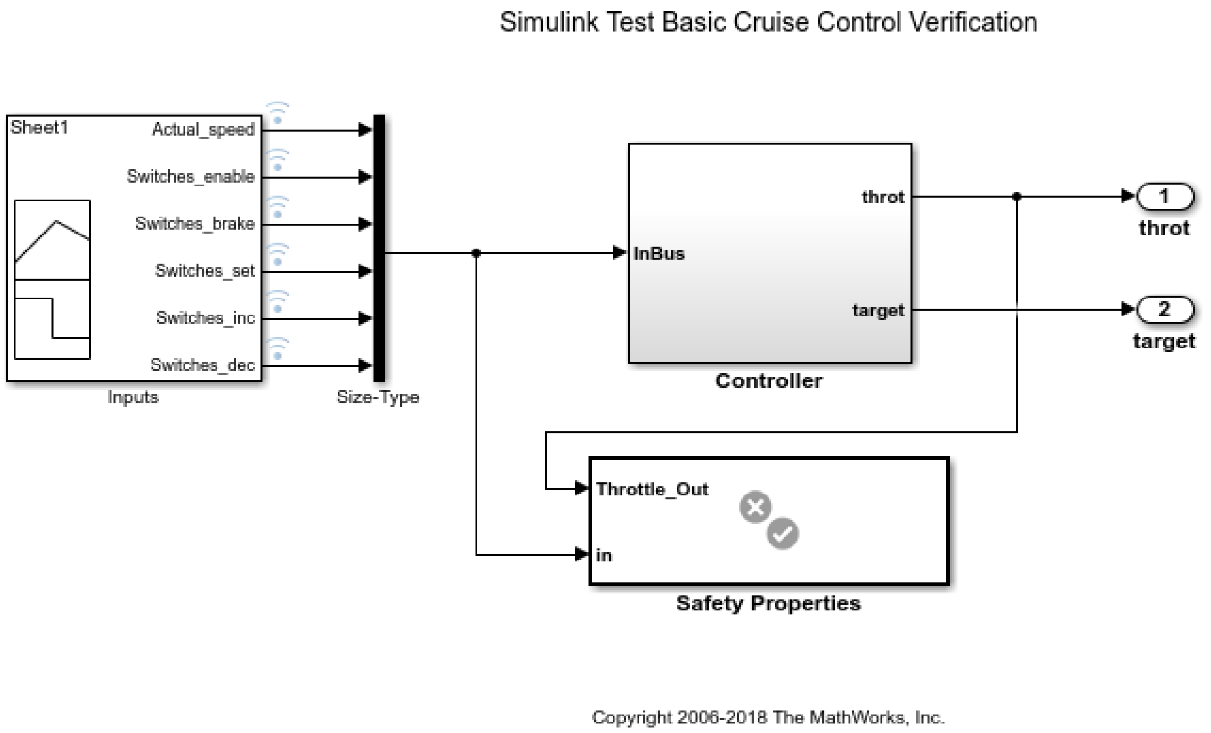
Import a Standalone Model as a Test Harness

This example shows how to import a standalone test model to create a test harness in Simulink Test.

The main model `sltestBasicCruiseControl` is a cruise control system, with root import and output blocks.


Simulink Test: Basic Cruise Control
1
InBus
InBus
throt
target
Controller
1
throt
2
target
Copyright 2006-2018 The MathWorks, Inc.

The test model contains a Signal Builder block driving a copy of the `Controller` subsystem, with a subsystem verifying that the throttle output goes to 0 if the brake is applied for three consecutive time steps.



Create a Test Harness from the Standalone Model

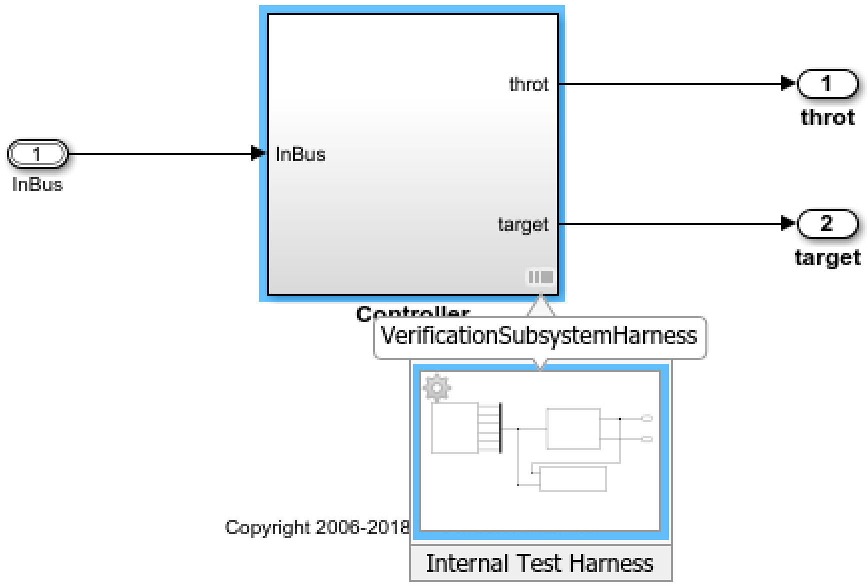
1. In the main model, right-click the `Controller` subsystem and select **Test Harness > Import for 'Controller'**.

2. Set the following harness properties:

- **Name**: `VerificationSubsystemHarness`
- **Simulink model to import:** Click **Browse** and select `sltestBasicCruiseControlHarnessModel` in the MATLAB® `examples/simulinktest` directory.
- **Component under Test in imported model**: `Controller`

3. Click **OK**.

A test harness is created from the standalone model, owned by the `Controller` subsystem in the main model. Click the badge to preview the test harness.



See Also

`sltest.harness.import`

Related Examples

- "Test Harness and Model Relationship" on page 2-2
- "Synchronize Changes Between Test Harness and Model" on page 2-53

Synchronize Changes Between Test Harness and Model

In this section...

A test harness provides an isolated environment to test design changes. You can synchronize changes from the test harness to the main model, or from the main model to the test harness. Synchronization includes these model elements:

- The component under test
- Block parameters
- Optionally, the model or test harness configuration set
- Optionally, the model workspace parameters

You do not need to synchronize base workspace data because it is available to both test harness and main model.

Set Synchronization for a New Test Harness

When creating a test harness, you specify when changes in the test harness are synchronized with the main model. Synchronization can occur automatically or manually. If you plan to try out different component designs in the test harness, use manual synchronization to avoid overwriting the component in the main model. Depending on the type of component under test in your harness, you can select from several synchronization options, which are combinations of the following actions:

- On harness open — The component in the test harness is updated when the harness opens. Synchronizing on harness open is useful if you update the design in the main model.
- On harness close — The component in the main model is updated when the harness closes. Synchronizing on harness close is useful if you make design changes in the test

harness. Avoid synchronizing on harness close if you want to prevent inadvertent changes to the component in the main model.

- During push — Synchronization occurs manually, by selecting **Analysis** > **Test Harness** > **Push Component and Parameters to Main Model**.
- During rebuild — Synchronization occurs manually, by selecting **Analysis** > **Test Harness** > **Rebuild Harness from Main Model**.

If you use the command line, set the `SynchronizationMode` property with `sltest.harness.create`.

Note If you create a test harness in SIL or PIL mode for a Model block, the block mode in the test harness is changed to SIL or PIL, respectively. This mode is not updated to the main model when you close the test harness.

Maintain SIL or PIL Block Fidelity If you use a software-in-the-loop (SIL) or processor-in-the-loop (PIL) block in the test harness, consider setting the test harness to rebuild every time it opens. Regularly rebuilding the test harness ensures that the generated code referenced by the SIL/PIL block reflects the main model.

Change Synchronization of an Existing Test Harness

To change a test harness synchronization mode:

1. Close the test harness.
2. In the main model, click the harness badge on the block or the Simulink canvas.
3. In the test harness thumbnail preview, click the **Harness operations** icon and select **Properties**.
4. Change the **Synchronization Mode** in the properties dialog box.

If you use the command line, set the `SynchronizationMode` property with `sltest.harness.set`.

Synchronize Configuration Set and Model Workspace Data

To synchronize the configuration set and workspace parameters between the test harness and main model, select **Update Configuration Parameters and Model Workspace data on rebuild** in the harness creation or harness properties dialog box.

Check for Unsynchronized Component Differences

If your test harness does not synchronize changes, you can check for unsynchronized component differences between the test harness and main model. Checking for unsynchronized differences can be useful if:

- You are making tentative design changes in the test harness and want to check that the main model component is not overwritten.
- You have made design changes to the main model and want to check which test harnesses must be rebuilt.

From the test harness window, select **Analysis > Test Harness > Check** to check for differences. If the component differs, you can push changes from the test harness to the main model, or rebuild the test harness from the main model. Also see the `sltest.harness.check` function.

Consider these conditions when checking for unsynchronized differences:

- `sltest.harness.check` only includes the block diagram, block parameters, and mask parameters in the comparison between the test harness and main model. Port options, compiled attributes, hidden parameters, and model reference data logging parameters are not included in the comparison.
- If the component contains a Simscape™ Solver Configuration block, the check result always shows that the component differs between the test harness and main model. The Solver Configuration block is affected by Simscape blocks outside the component, and therefore always differs between the test harness and main model.

Rebuild a Test Harness

Rebuild a test harness to reflect the latest state of the main model. In the test harness, select **Analysis > Test Harness > Rebuild Harness from Main Model**. In addition to updating the component under test and block parameters, this operation rebuilds harness conversion subsystems. If the test harness does not have conversion subsystems, rebuilding adds them.

Rebuilding can disconnect signal lines. For example, if signal names changed in the main model, signal lines in the test harness can be disconnected. If lines are disconnected, reconnect signal lines to the component under test or conversion subsystems.

For more information, see “Create Test Harnesses and Select Properties” on page 2-13 and `sltest.harness.rebuild`.

Push Changes from Test Harness to Model

After changing your system in the test harness, you can push changes to the main model. In the test harness, select **Analysis > Test Harness > Push Component and Parameters to Main Model**. This process overwrites the component in the main model.

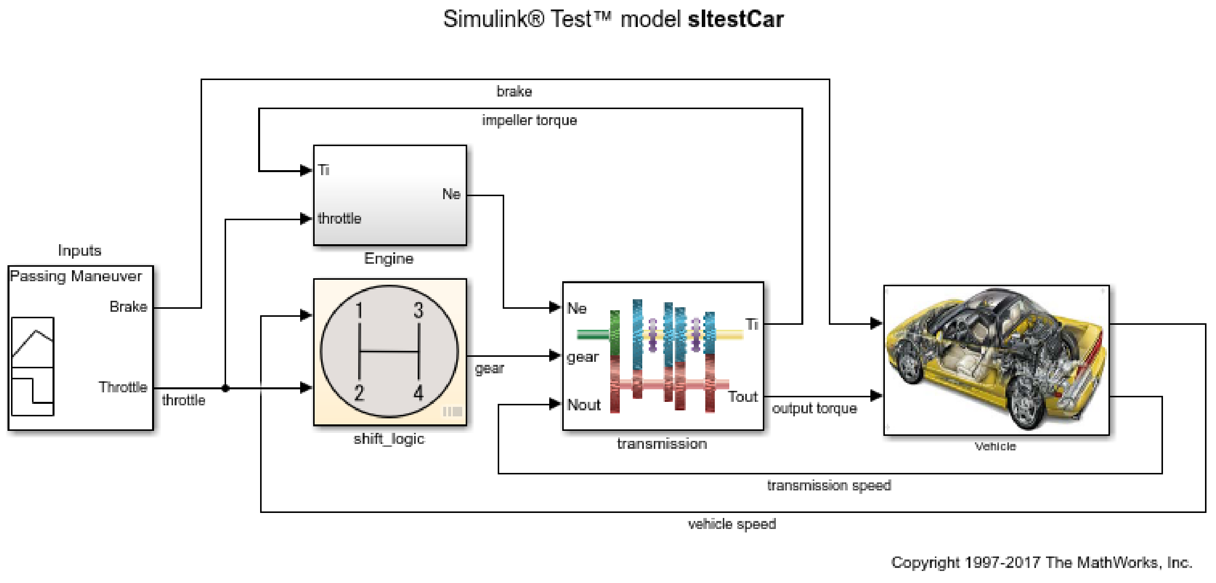
Check Component and Push Parameter to Main Model

This example shows a basic workflow of updating a parameter in a test harness, checking the synchronization between the test harness and main model, and pushing the parameter change from the test harness to the main model.

This example also includes programmatic steps.

Open the model `sltestCar`. The model includes a transmission shift controller algorithm and simplified powertrain and vehicle dynamics.

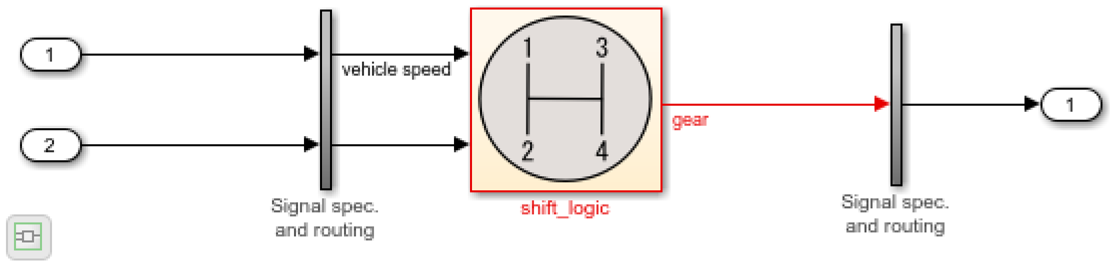
```
open_system('sltestCar');
```



Update the Mask Parameter in the Test Harness

1. Open the test harness. Click the badge on the `shift_logic` chart and select the `ShiftLogic_InportHarness` test harness. The test harness is set to synchronize only when you push to or rebuild from the main model.

```
sltest.harness.open('sltestCar/shift_logic','ShiftLogic_InportHarness');
```



2. Double-click the `shift_logic` subsystem. For **Delay before gear change (tick)**, enter 4. Click **OK**.

```
shiftLogicMask = Simulink.Mask.get('ShiftLogic_InportHarness/shift_logic');
maskParamValue = shiftLogicMask.Parameters.Value;
shiftLogicMask.Parameters.Value = '4';   % Set the new parameter value
```

Check Synchronization between Test Harness and Main Model

On the command line, run the `sltest.harness.check` function.

```
[comparison,details] = sltest.harness.check('sltestCar/shift_logic',...
    'ShiftLogic_InportHarness');
```

The results show that the component under test is different in the test harness due to the updated mask parameter.

```
comparison
```

```
comparison =

  logical

   0
```

```
details
```

```
details =

  struct with fields:

     overall: 0
    contents: 1
      reason: 'The contents of harnessed component and the contents of the component in
```

Update the Parameter to the Main Model

1. In the test harness, select **Analysis > Test Harness > Push Component and Parameters to Main Model.**

2. In the main model, double-click the shift_logic subsystem. The parameter value is updated.

```
sltest.harness.push('sltestCar/shift_logic','ShiftLogic_InportHarness')
```

Re-check Synchronization between Test Harness and Main Model

On the command line, update the main model and test harness. Then, run the `sltest.harness.check` function.

```
set_param('sltestCar','SimulationCommand','update');
set_param('ShiftLogic_InportHarness','SimulationCommand','update');

[comparison,details] = sltest.harness.check('sltestCar/shift_logic',...
    'ShiftLogic_InportHarness');
```

The results show that the component under test is the same between the test harness and the main model.

```
comparison
```

```
comparison =

  logical

   1
```

```
details
```

```
details =

  struct with fields:

     overall: 1
    contents: 1
      reason: 'The checksum of the harnessed component and the component in the main mo
```

```
close_system('sltestCar',0);
```

See Also

sltest.harness.check | sltest.harness.push | sltest.harness.rebuild

Related Examples

- “SIL Verification for a Subsystem” on page 5-2

Test Library Blocks

In this section...

If your model includes instances of blocks from a library, you can test both the source block in the library, and individual block instances in other models. First, create test harnesses for a library block to test your design. Once the library block meets your requirements, create test harnesses for linked blocks and test the subsystem instances. You can move test harnesses from the library to an instance and an instance to the library.

Library Testing Workflow

This procedure outlines an example workflow for testing library subsystems and linked subsystems.

1. Create a test case and a test harness for the library subsystem.
2. Test the library subsystem. If it fails your requirements, revise the design and test again.
3. Lock the library when your tests pass.
4. In your model, create a linked subsystem and retain the library test harnesses.
5. Compare the output of the linked instance to that of the library block using an equivalence test case.
6. Create additional test cases and test harnesses for the linked instance.
7. Promote a test harness from the linked subsystem to the library if you want to include the test harness with future linked subsystems.

Library and Linked Subsystem Test Harnesses

A test harness for a library subsystem has specific properties:

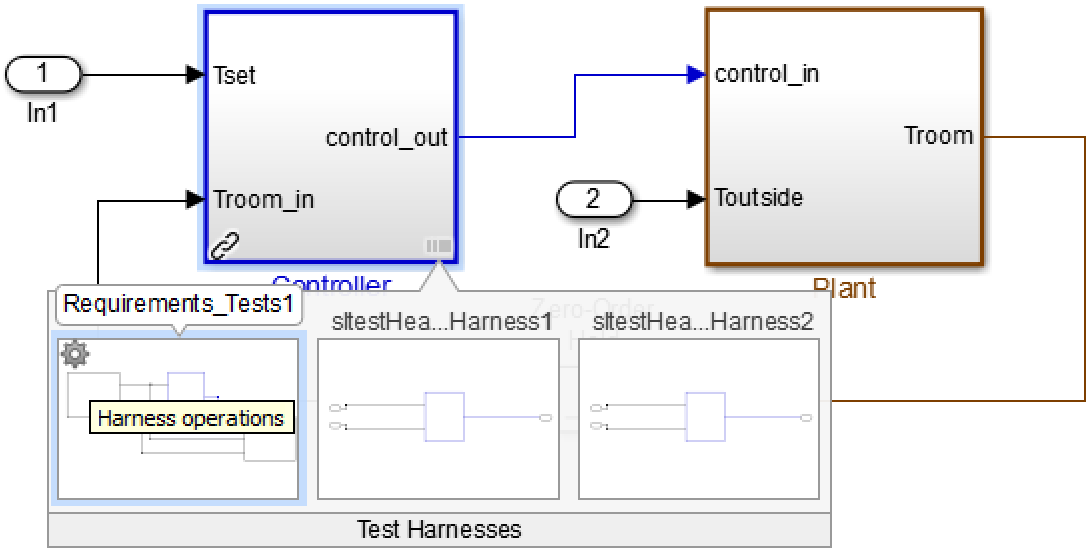
- Libraries do not compile, so a test harness for a library subsystem does not use compiled attributes such as data type or sample rate.

- A test harness for a library subsystem does not generate conversion subsystems for the block inputs and outputs.
- A library subsystem test harness does not use push or rebuild operations, because libraries do not use configuration parameters.

When you create a linked subsystem from a library subsystem, test harnesses copy to the linked instance. If you do not need the test harnesses, you can delete them. For instructions on deleting all test harnesses from a model, see “Manage Test Harnesses” on page 2-28.

When you create a test harness for a linked subsystem, the harness associates with the linked subsystem, not the library subsystem. You can move a test harness from a linked subsystem to the library subsystem. For example, this linked subsystem `Controller` has three test harnesses. To move the `Requirements_Tests1` test harness to the library:

1. Click the harness badge on the linked subsystem.
2. Click the **Harness Operations** icon.



3. Select **Move to Library**.
4. A dialog box informs you that moving the harness removes it from the linked subsystem.

5 After confirmation, the harness appears with the library subsystem.

Edit Library Block from a Test Harness

You can apply an iterative design and test workflow to libraries by testing a library block in a test harness and updating the component under test. Changes to the component under test synchronize to the library when you close the test harness.

If you have a library block whose design is complete, set your test harnesses to prevent changes to the component under test. You can set this property when you create the test harness or after harness creation. See “Create Test Harnesses and Select Properties” on page 2-13.

Testing a Library and a Linked Block

Verify a reusable subsystem in a library and in a larger system.

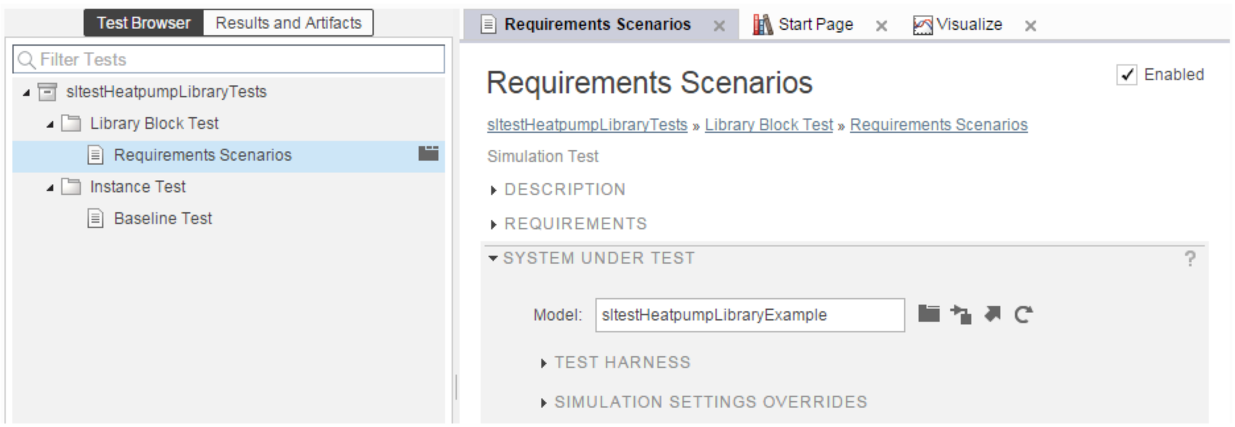
This example demonstrates a test case that confirms a library block meets a short set of requirements. After testing the library block, you execute a baseline test of a linked block and capture the baseline results. You then promote the baseline test harness to the library.

The library block controls a simple heatpump system by supplying on/off signals to a fan and compressor, and specifying the heatpump mode (heating or cooling).

Open the Test File

Enter the following to store paths and filenames for the example, and to open the test file. The test file contains a test case for the library block and for the block instance in a closed-loop model.

```
filePath = fullfile(matlabroot,'toolbox','simulinktest','simulinktestdemos');
testFile = 'sltestHeatpumpLibraryTests.mldatx';
library = 'sltestHeatpumpLibraryExample';
system = 'sltestHeatpumpLibraryLinkExample';
open(fullfile(filePath,testFile));
```



Requirements Scenarios

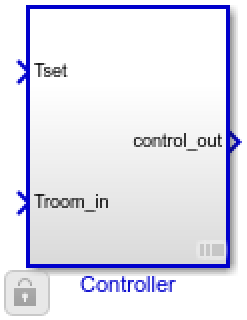
sltestHeatpumpLibraryTests » Library Block Test » Requirements Scenarios

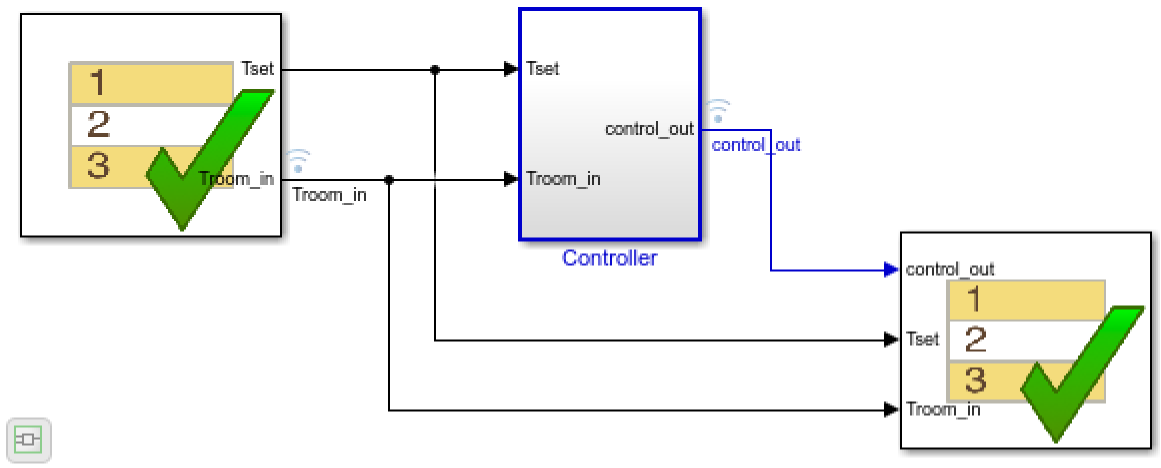
Simulation Test

Model: sltestHeatpumpLibraryExample

Expand the **Library Block Test** test suite, and highlight the **Requirements Scenarios** test case in the test browser. Expand the **Test Harness** section of **System Under Test**, and click the arrow to open the test harness for the library block.

```
open_system(library);
sltest.harness.open([library '/Controller'],'Requirements_Tests');
```





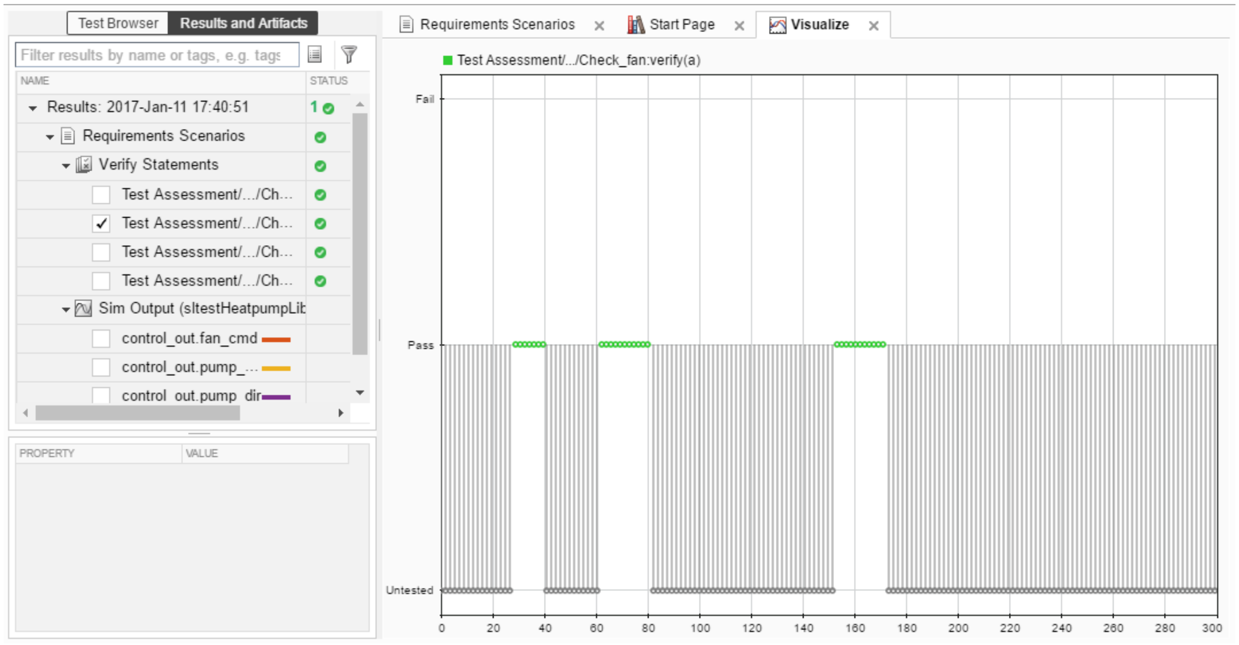
The Test Sequence block sets three scenarios for the controller:

- The controller at idle
- The controller activating the fan only
- The controller activating the heating anc cooling system

The Test Assessment block in the test harness checks the signals for each scenario. Since the test inputs and assessments are contained in the test harness, and no baseline data is being captured, the test case is a simulation test.

Run the Requirements-Based Test

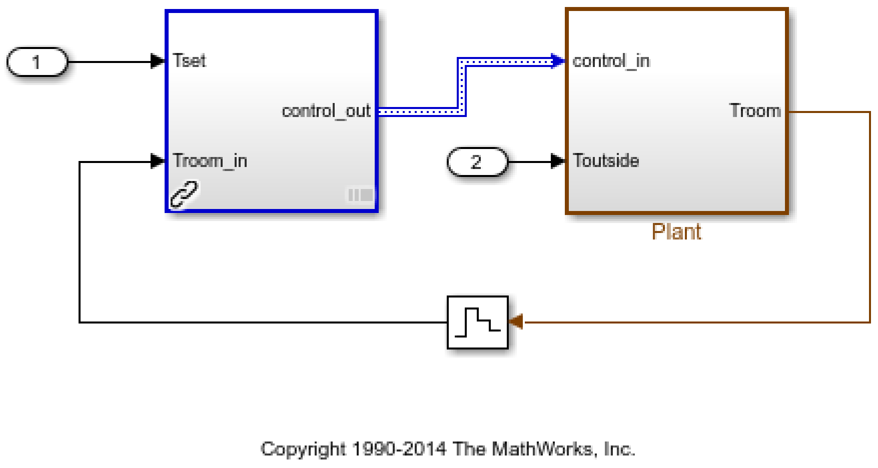
In the Test Manager, run the Requirements Scenarios test case. The `verify` statement results show that the `control_out` signals pass.



Open the Linked Block Model

In the Test Manager, expand **Instance Test**. Highlight the **Baseline Test** test case. In the **System Under Test**, click the arrow next to the **Model** field to open the model.

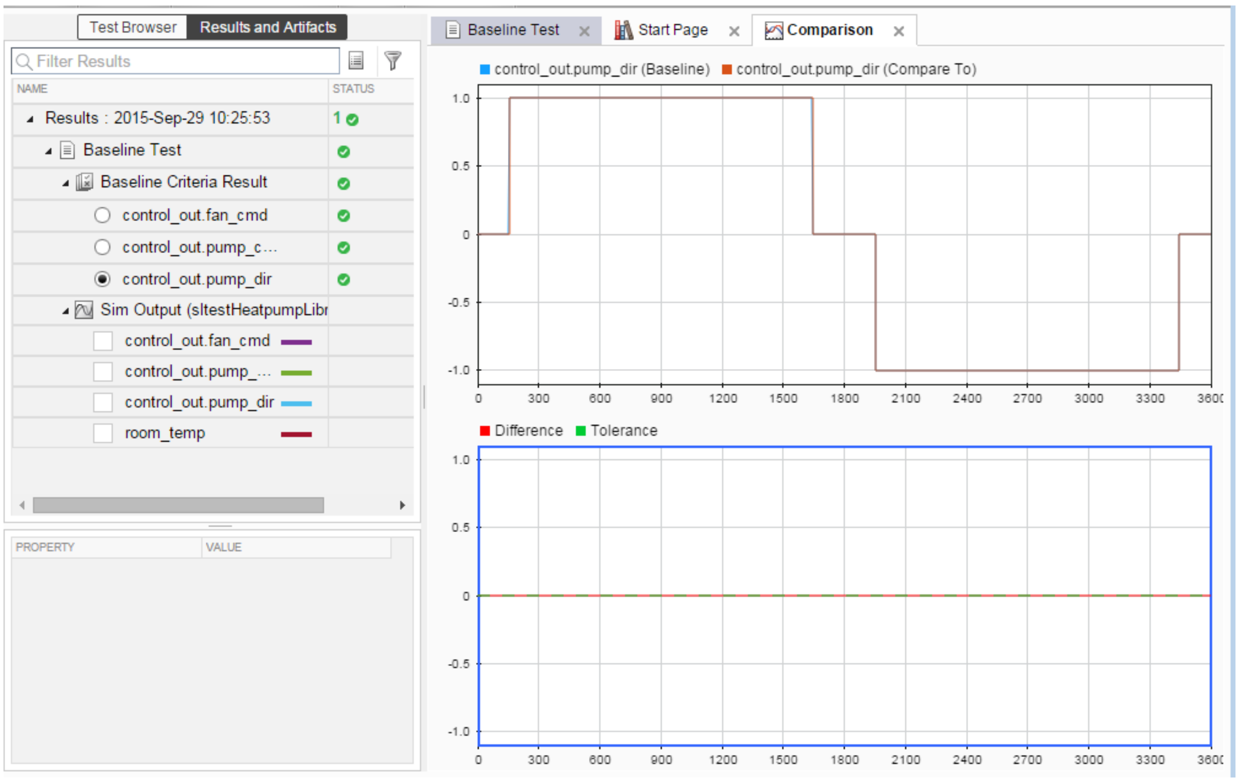
```
sltest.harness.close([library '/Controller'], 'Requirements_Tests');
open_system(system);
sim(system);
```



The controller is a linked block to the library. It is associated with a test harness **Baseline Test** that compares simulation results of the instance against baseline data. In your workflow, successful baseline testing for an instances of a library block can show that the linked block simulates correctly in the containing model. The test harness supplies a sine wave temperature and captures the controller output.

Run the Baseline Test and Observe Results

In the Test Manager, click **Run** to execute the test. The results show that the baseline test passes.



Move the Test Harness to the Library

If you develop a particularly useful test for a linked block, you can promote the test harness from a linked block to the source library block. The test harness then copies to all future instances of the library block.

Move the **Baseline_controller_tests** test harness to the library block:

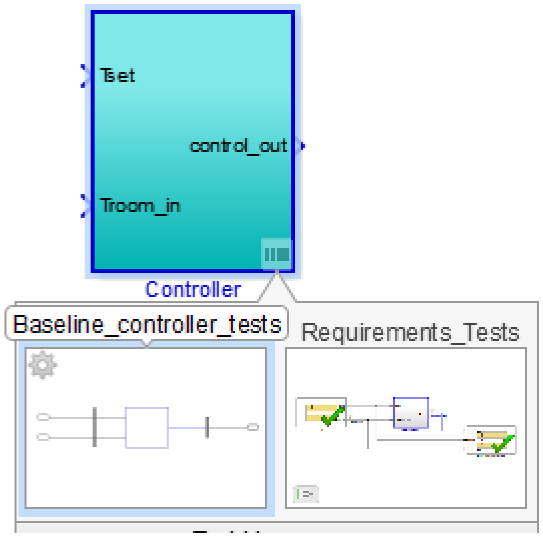
1. In the sltestHeatpumpLibraryLinkExample model, click the harness badge and hover over the **Baseline_controller_tests** test harness.

2. Click the harness operations icon

⚙

3. Select **Move to Library**. A dialog informs you that the operation deletes the test harness from the instance and adds it to the library. Click **Yes**.

4. The test harness moves to the Controller library block.



```
close_system(library,0);
close_system(system,0);
clear(filePath,library,system,testFile);
sltest.testmanager.clear;
sltest.testmanager.clearResults;
```

See Also

Related Examples

- “Testing a Library and a Linked Block”

Test Sequences and Assessments

Test Sequence Basics

In this section...

A test sequence consists of test steps arranged in a hierarchy. You can use a test sequence to define test inputs and to define how a test will progress in response to the simulation. A test step contains actions that execute at the beginning of the step. A test step can contain transitions that define when the step stops executing, and which test step executes next. Actions and transitions use MATLAB as the action language. You create test sequences by using the Test Sequence block and the Test Sequence Editor on page 3-24.

Test Sequence Hierarchy

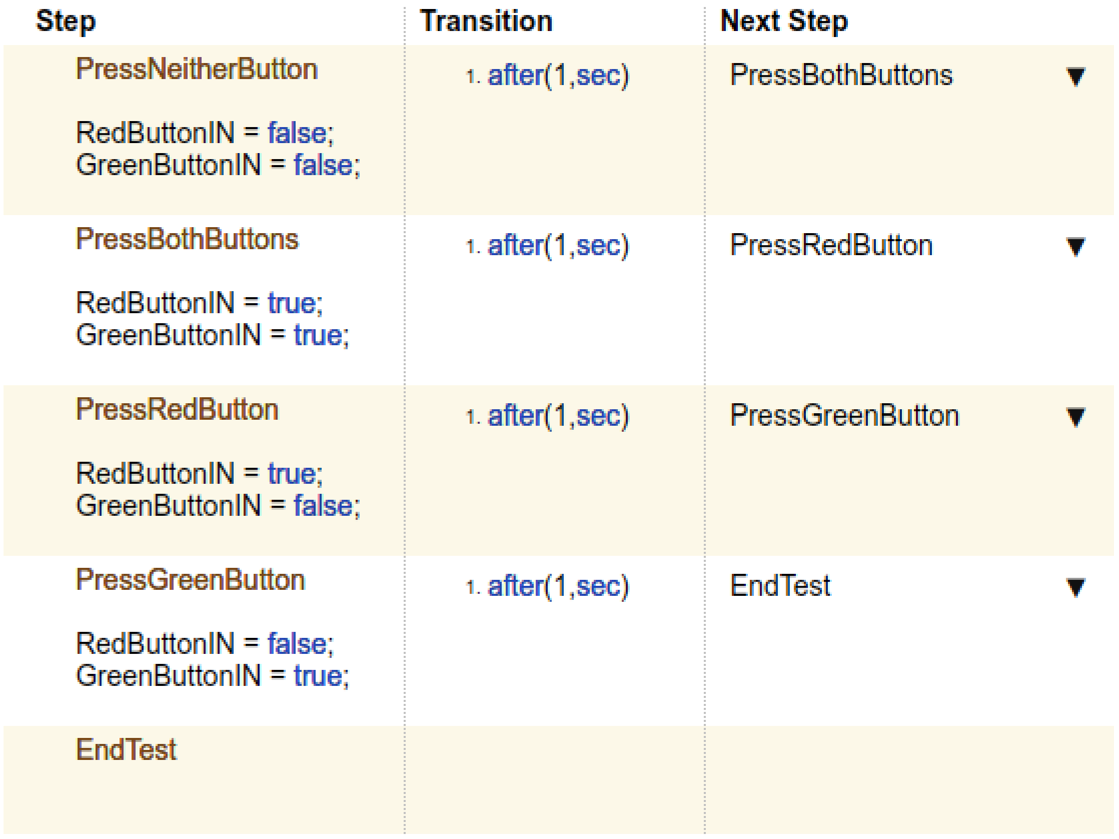
Test sequences can have parent steps and substeps. Substeps can activate only if the parent step is active. A group of steps in the same hierarchy level shares a common transition type. When you create a test step, the step becomes a transition option for other steps in the same group.

Transition Types

Test sequences transition from one step to another in two ways:

- **Standard transition:** You can define a sequence of actions that react to simulation conditions using a standard transition sequence. Standard transition sequences start with the first step and progress according to transition conditions and next steps.

  This test sequence sets the value of Boolean outputs `RedButtonIn` and `GreenButtonIn`, with transitions happening after each step has been active for 1 sec.



Step	Transition	Next Step

PressNeitherButton
 	
RedButtonIN = false;
GreenButtonIN = false;
	1. after(1,sec)	PressBothButtons ▼

PressBothButtons
 	
RedButtonIN = true;
GreenButtonIN = true;
	1. after(1,sec)	PressRedButton ▼

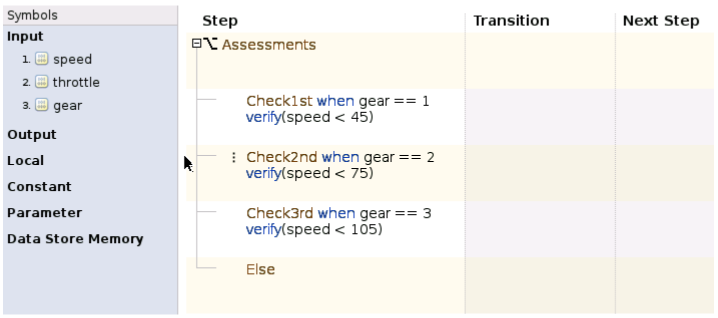
PressRedButton
 	
RedButtonIN = true;
GreenButtonIN = false;
	1. after(1,sec)	PressGreenButton ▼

PressGreenButton
 	
RedButtonIN = false;
GreenButtonIN = true;
	1. after(1,sec)	EndTest ▼

EndTest		

- **When decomposition:** `When` decomposition sequences are analogous to switch statements in programming. Your sequence can act based on specific conditions occurring in your model. In a `When` decomposition sequence, steps activate based on a condition that you define after the step name. Transitions are not used between steps.

  This When decomposition contains three `verify` statements. Each `verify` statement is active when the signal `gear` is equal to a different value. For more information, see “Test Sequence Editor” on page 3-24.



Symbols
Input
1. speed
2. throttle
3. gear
Output
Local
Constant
Parameter
Data Store Memory

Step	Transition	Next Step
Assessments		
Check1st when gear == 1 verify(speed < 45)		
Check2nd when gear == 2 verify(speed < 75)		
Check3rd when gear == 3 verify(speed < 105)		
Else		

Create a Basic Test Sequence

In this example, you create a simple test sequence for a transmission shift logic controller.

1 Open the model. At the command line, enter

```
openExample('simulinktest/TransmissionDownshiftTestSequenceExample')
```

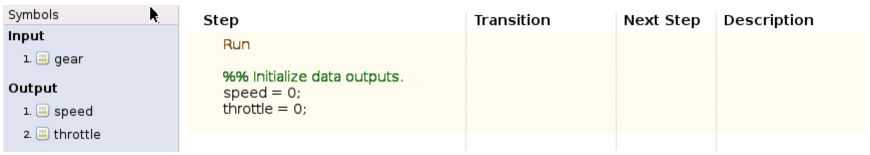
2 Right-click the `shift_controller` subsystem and select **Test Harness** > **Create for 'shift_controller'**.

3 In the Create Test Harness dialog box, under **Sources and Sinks**:

- Under **Sources and Sinks**, select `Test Sequence` from the source drop-down menu.
- Under **Sources and Sinks**, select **Add separate assessment block**.
- Select **Open harness after creation**.

4 Click **OK**. The test harness for the `shift_controller` subsystem opens.

5 Double-click the Test Sequence block. The Test Sequence Editor opens.



Symbols
Input
1. gear
Output
1. speed
2. throttle

Step	Transition	Next Step	Description

Run
 	
%% Initialize data outputs.
speed = 0;
throttle = 0;			

6 Create the test sequence.

a Rename the first step `Accelerate` and add the step actions:

```
speed = 10*ramp(et);
throttle = 100;
```

b Right-click the `Accelerate` step and select **Add step after**. Rename this step `Stop`, and add the step actions:

```
throttle = 0;
speed = 0;
```

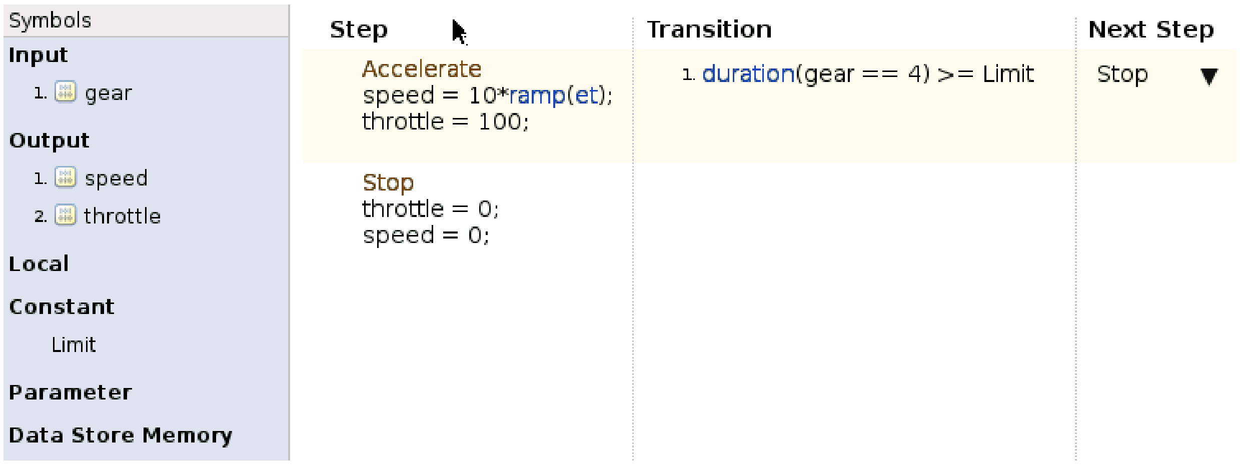
c Enter the transition condition for the `Accelerate` step. In this example, `Accelerate` transitions to `Stop` when the system is in fourth gear for 2 seconds. In the **Transition** column, enter:

```
duration(gear == 4) >= Limit
```

In the **Next Step** column, select `Stop`.

d Add a constant to define `Limit`. In the **Symbols** pane, hover over **Constant** and click the add data button. Enter `Limit` for the constant name.

e Hover over `Limit` and click the edit button. In the **Constant value** field, enter 2. Click **OK**.



Step	Transition	Next Step

Accelerate
speed = 10*ramp(et);
throttle = 100;
	1. duration(gear == 4) >= Limit	Stop

Stop
throttle = 0;
speed = 0;		

Create Basic Test Assessments

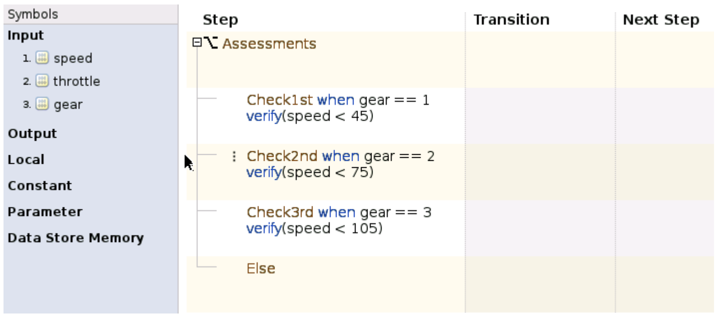
1 Continuing the example, in the test harness, double-click the Test Assessment block to open the editor. The editor displays a When decomposition sequence.

2 Rename the first step `Assessments`.

3 Add two steps to `Assessments`. Right-click the `Assessments` step and select **Add sub-step**. Do this a second time. There should be four steps under `Assessments`.

4 Enter the names and actions for the four substeps.

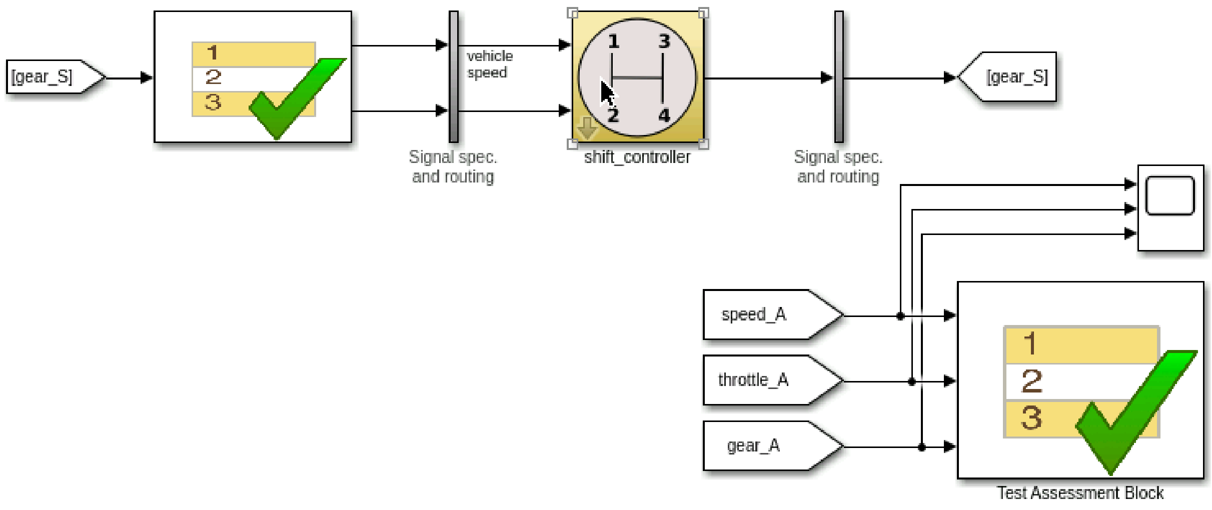
```
Check1st when gear == 1
verify(speed < 45)

Check2nd when gear == 2
verify(speed < 75)

Check3rd when gear == 3
verify(speed < 105)

Else
```



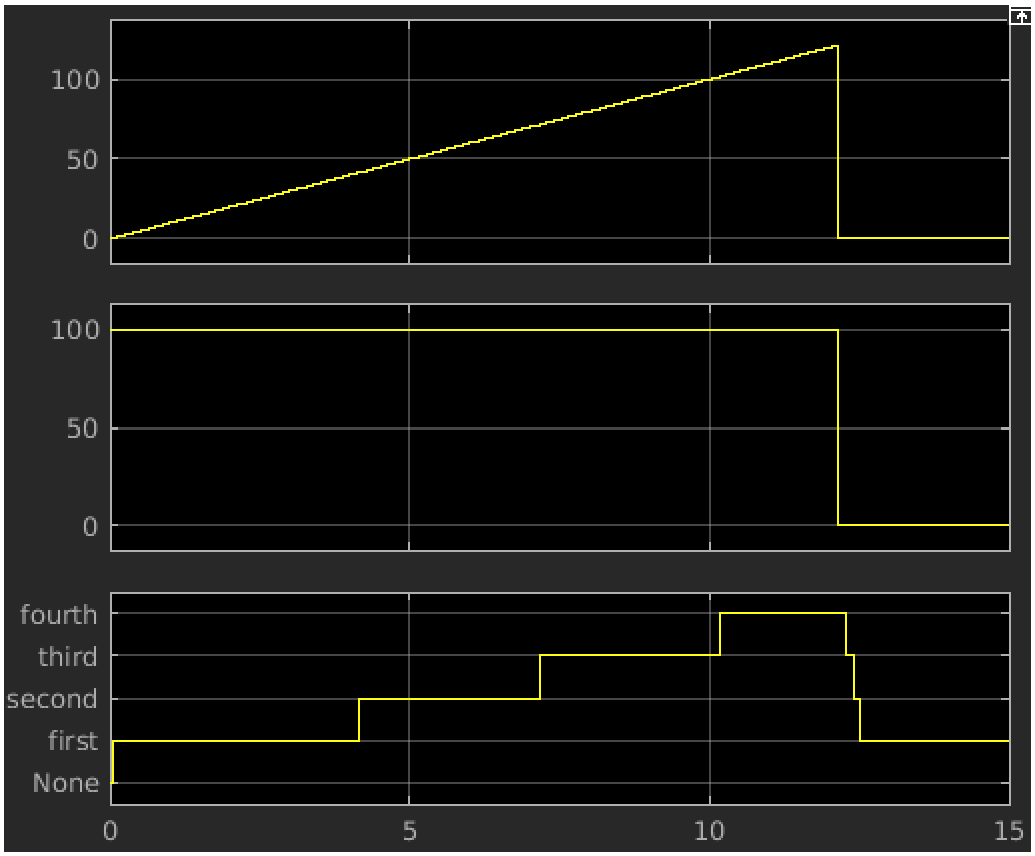
Step	Transition	Next Step
Assessments		
Check1st when gear == 1 verify(speed < 45)		
Check2nd when gear == 2 verify(speed < 75)		
Check3rd when gear == 3 verify(speed < 105)		
Else		

The fourth step `Else` has no actions. `Else` handles simulation conditions outside of the preceding `when` conditions.

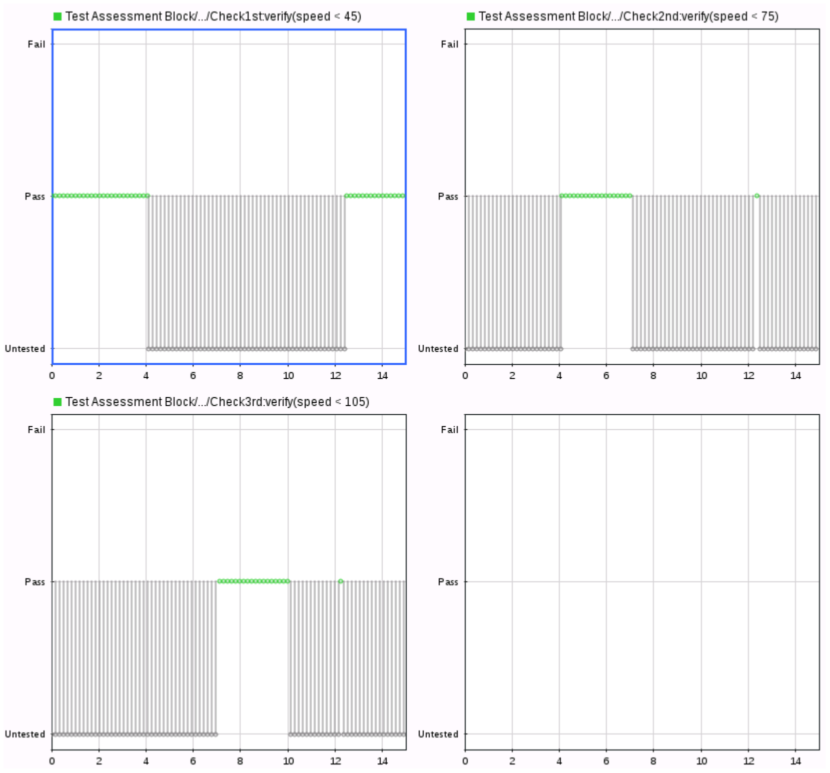
5 Add a scope to the harness and connect the `speed`, `throttle`, and `gear` signals to the scope.



6 Set the model simulation time to 15 seconds and simulate the test harness. View the signal data by opening the scope.



7 View the results of the `verify` statements in the Simulation Data Inspector.



See Also

“Test Sequence Editor” on page 3-24 | “Test Sequence and Assessment Syntax” on page 3-60 | Test Sequence

Assess Simulation and Compare Output Data

In this section...

Overview

Functional testing requires assessing simulation behavior and comparing simulation output to expected output. For example, you can:

- Analyze signal behavior in a time interval after an event.
- Compare two variables during simulation.
- Compare timeseries data to a baseline.
- Find peaks in timeseries data, and compare the peaks to a pattern.

This topic provides an overview to help you author assessments for your particular application. In the topic, you can find links to more detailed examples of each assessment.

You can include assessments in a test case, a model, or a test harness.

- In a test case, you can:
  - Compare simulation output to baseline data.
  - Compare the output of two simulations.
  - Post-process simulation output using a custom script.
- In a test harness or model, you can:
  - Verify logical conditions in run-time using a `verify` statement, which returns a `pass`, `fail`, or `untested` result for each time step.
  - Use `assert` statements to stop simulation on a failure.
  - Use blocks from the Model Verification or Simulink Design Verifier library.

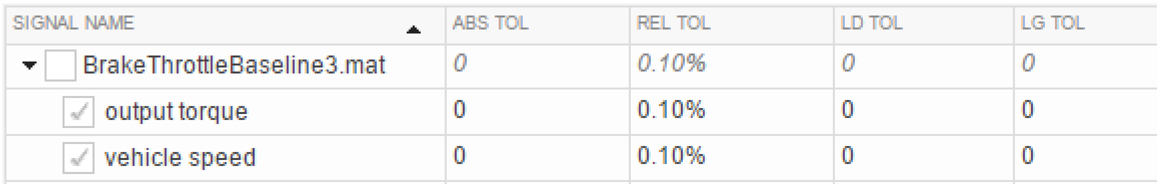
Compare Simulation Data to Baseline Data or Another Simulation

Baseline criteria are tolerances for simulation data compared to baseline data. Equivalence criteria are tolerances for two sets of simulation data, each from a different simulation. You can set tolerances for numeric, enumerated, or logical data.

Set a numeric tolerance using absolute or relative tolerances. Set time tolerances using leading and lagging tolerances. For numeric data, you can specify absolute tolerance, relative tolerance, leading tolerance, or lagging tolerance. For enumerated or logical data, you can specify leading or lagging tolerance. Results outside the tolerances fail. For more information, see “Set Signal Tolerances” on page 7-90.

Specify the baseline data and tolerances in the Test Manager **Baseline Criteria** or **Equivalence Criteria** section. Results appear in the **Results and Artifacts** pane. The comparison plot displays the data and differences.

This graphic shows an example of baseline criteria. The baseline criteria sets a relative tolerance for signals `output torque` and `vehicle speed`.



SIGNAL NAME	ABS TOL	REL TOL	LD TOL	LG TOL
BrakeThrottleBaseline3.mat	*0*	*0.10%*	*0*	*0*
✓ output torque	0	0.10%	0	0
✓ vehicle speed	0	0.10%	0	0

Post-Process Results With a Custom Script

You can analyze simulation data using specialized functions by using a custom criteria script. For example, you could find peaks in timeseries data using Curve Fitting Toolbox™ functions. A custom criteria script is MATLAB code that runs after the simulation. Custom criteria scripts use the MATLAB Unit Test framework.

Write a custom criteria script in the Test Manager **Custom Criteria** section of the test case. Custom criteria results appear in the **Results and Artifacts** pane. Results are shown for individual MATLAB Unit Test qualifications. For more information, see “Process Test Results with Custom Scripts” on page 7-111.

This simple test case custom criteria verifies that the value of `slope` is greater than 0.

```
% A simple custom criteria
test.verifyGreaterThan(slope,0,'slope must be greater than 0')
```

Run-Time Assessments

verify Statements

For general run-time assessments, use `verify` statements. A `verify` statement evaluates a logical expression and returns a pass, fail, or untested result for each simulation time step. `verify` statements can include temporal and conditional syntax. A failure does not stop simulation.

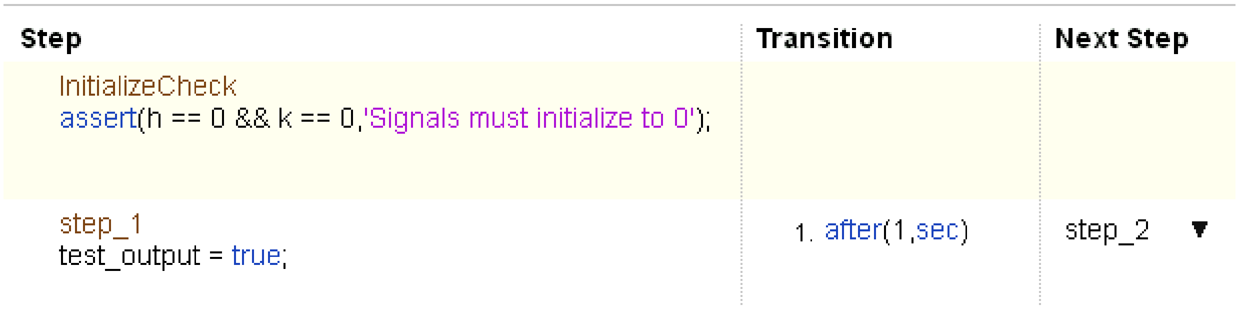
Enter `verify` statements in a Test Assessment or Test Sequence block, using the Test Sequence Editor. You can use `verify` statements with or without a test case in the Test Manager. Without a test case, results appear in the Simulation Data Inspector. With a test case, results appear in the Test Manager.

For information on using `verify` statements in your model, see “Assess Model Simulation Using verify Statements” on page 3-15.

assert Statements

You can use `assert` statements in a Test Assessment or Test Sequence block to stop executing an invalid test. `assert` evaluates a logical argument, but unlike `verify`, `assert` stops simulation. Failures appear as simulation errors. To make results easier to interpret, add an optional message.

For example, if a component under test outputs two signals `h` and `k`, and the test requires `h` and `k` to initialize to `0`, use `assert` to stop the test if the signals do not initialize. This `assert` statement returns a message `'Signals must initialize to 0'` if the logical condition `h == 0 && k == 0` fails.



Step	Transition	Next Step
InitializeCheck assert(h == 0 && k == 0,'Signals must initialize to 0');		
step_1 test_output = true;	1. after(1,sec)	step_2 ▼

Assessments for Real-Time Testing

If you are using a real-time test case, or if you want to reuse a desktop simulation test case on a real-time target, use `verify` statements. `verify` statements are built into the real-time application, and run on the real-time target. See “Assess Model Simulation Using verify Statements” on page 3-15.

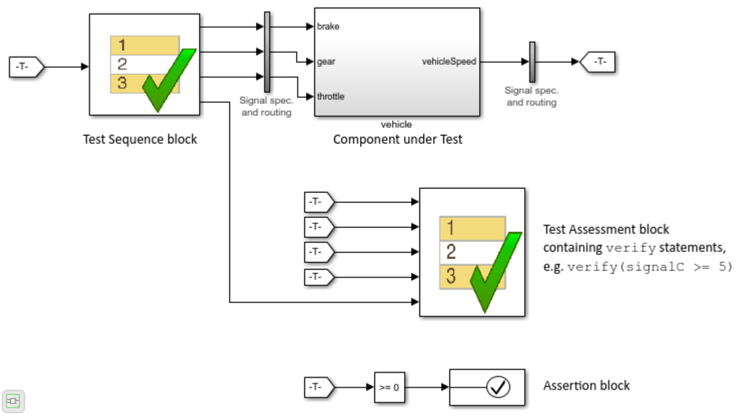
Model Verification Blocks

Use blocks from the Simulink Model Verification library or the Simulink Design Verifier library to assess signals in your model or test harness. `pass`, `fail`, or `untested` results from each block appear in the Test Manager. For more information, see “View Graphical Results From Model Verification Library” on page 3-79.

Examples of Run-Time Assessments

This example test harness includes:

- A `verify` statement in the Test Assessment block, verifying that `signalC >= 5`.
- An Assertion block verifying that `throttle >= 0`.



See Also

Related Examples

- “Compare Model Output To Baseline Data” on page 7-9
- “Test Two Simulations for Equivalence”

Assess Model Simulation Using verify Statements

You can verify model simulation by including a Test Assessment block in your model or test harness, and authoring `verify` statements in the Test Assessment block. `verify` statements return `pass`, `fail`, or `untested` results for both the overall simulation and individual time steps. Results appear in the Test Manager.

Activate verify Statements in the Test Assessment Block

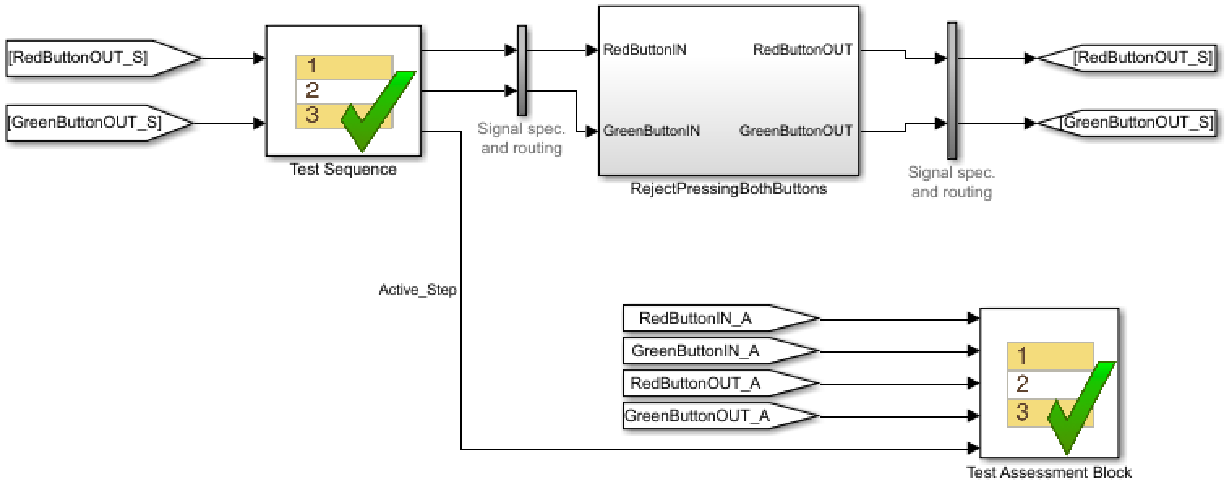
The Test Assessment contains a `When` decomposition on page 3-29 sequence. The `When` decomposition sequence helps you clearly define the simulation condition that activates each `verify` statement:

1. If your model uses a Test Sequence block source, consider activating each `verify` statement using the active Test Sequence block step.
2. If your model does not use a Test Sequence block source, or your test sequence steps do not correspond with conditions to verify, activate each `verify` statement using a signal condition.

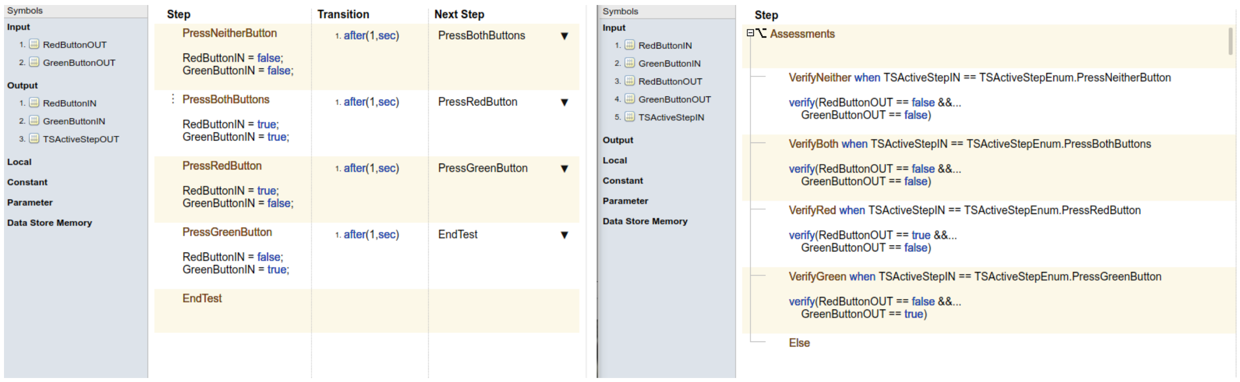
Activate verify Statements with Test Sequence Steps

Connect the Test Sequence and Test Assessment block with the active step signal from the Test Sequence block. Activate each `verify` statement with the active step.

For example, this test harness contains a Test Sequence and Test Assessment block. The blocks are connected by the `Active_Step` signal.



The Test Assessment block contains a `When` decomposition sequence with four substeps. Each contains a `verify` statement and is activated with a different Test Sequence block step.



Symbols
Input
1. RedButtonOUT
2. GreenButtonOUT
Output
1. RedButtonIN
2. GreenButtonIN
3. TSActiveStepOUT
Local
Constant
Parameter
Data Store Memory

Step	Transition	Next Step

PressNeitherButton
RedButtonIN = false;
GreenButtonIN = false;
	1. after(1,sec)	PressBothButtons

PressBothButtons
RedButtonIN = true;
GreenButtonIN = true;
	1. after(1,sec)	PressRedButton

PressRedButton
RedButtonIN = true;
GreenButtonIN = false;
	1. after(1,sec)	PressGreenButton

PressGreenButton
RedButtonIN = false;
GreenButtonIN = true;
	1. after(1,sec)	EndTest

EndTest		

Symbols
Input
1. RedButtonIN
2. GreenButtonIN
3. RedButtonOUT
4. GreenButtonOUT
5. TSActiveStepIN
Output
Local
Constant
Parameter
Data Store Memory

Step
Assessments
VerifyNeither when TSActiveStepIN == TSActiveStepEnum.PressNeitherButton verify(RedButtonOUT == false &&... GreenButtonOUT == false)
VerifyBoth when TSActiveStepIN == TSActiveStepEnum.PressBothButtons verify(RedButtonOUT == false &&... GreenButtonOUT == false)
VerifyRed when TSActiveStepIN == TSActiveStepEnum.PressRedButton verify(RedButtonOUT == true &&... GreenButtonOUT == false)
VerifyGreen when TSActiveStepIN == TSActiveStepEnum.PressGreenButton verify(RedButtonOUT == false &&... GreenButtonOUT == true)
Else

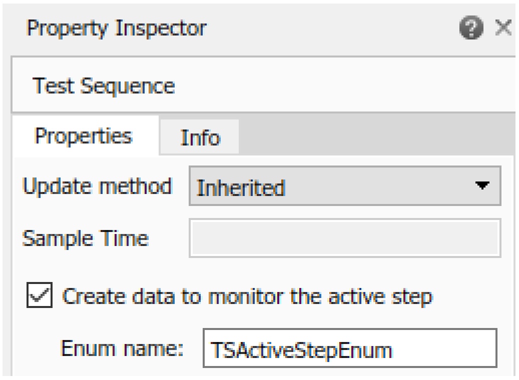
To activate verify statements in a Test Assessment with active steps in a Test Sequence blocks:

1 Create active step data output for the Test Sequence block:

  a Select the Test Sequence block.

   b Create a new enumerated data output. In the Property Inspector, select **Create data to monitor the active step**.

   c Name the enumeration.



Property Inspector

Test Sequence

Properties | Info

Update method: Inherited

Sample Time:

☑ Create data to monitor the active step

Enum name: TSActiveStepEnum

2 Create a data input for the Test Assessment block:

   a Open the Test Assessment block.

   b In the **Symbols** sidebar, next to **Input**, click the **Add data** icon.

   c Name the input.

3 In the block diagram, connect the Test Sequence block output to the Test Assessment block input.

4 Create a `When` decomposition sequence in the Test Assessment block.

   a The Test Assessment block is configured by default with a `When` decomposition sequence. To change between a standard sequence and a `When` decomposition sequence, right-click the parent step and select **When decomposition**.

   b For each `When` decomposition step, define when the step is active by using the active step enumeration data. For example:

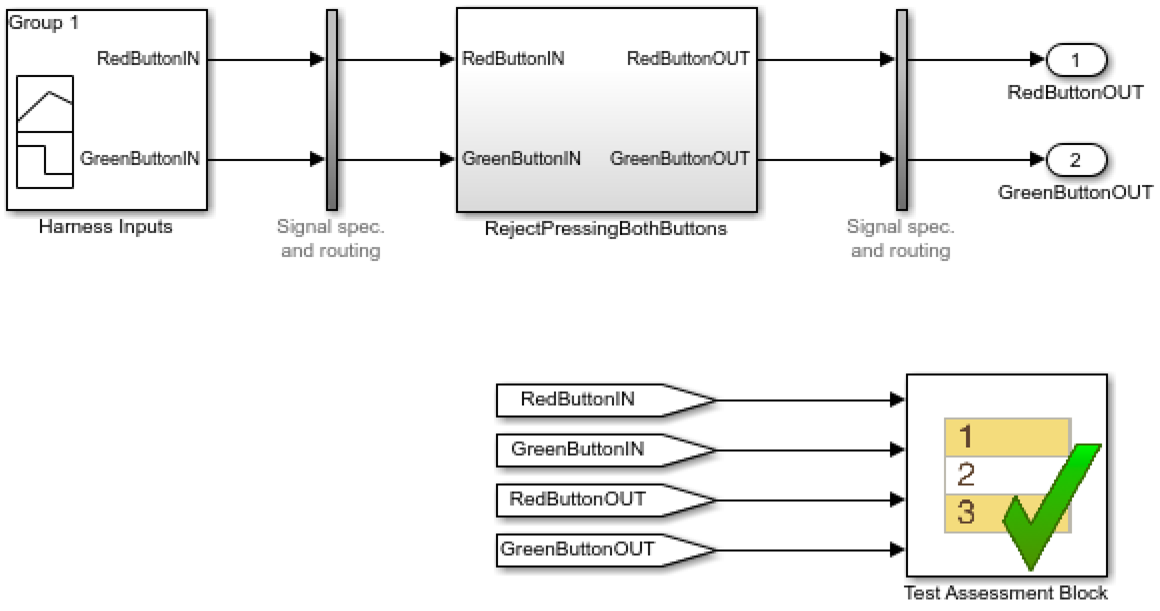
```
VerifyBoth when TSActiveStepIN == TSActiveStepEnum.PressBothButtons
```

   c Add `verify` statements to each assessment step.

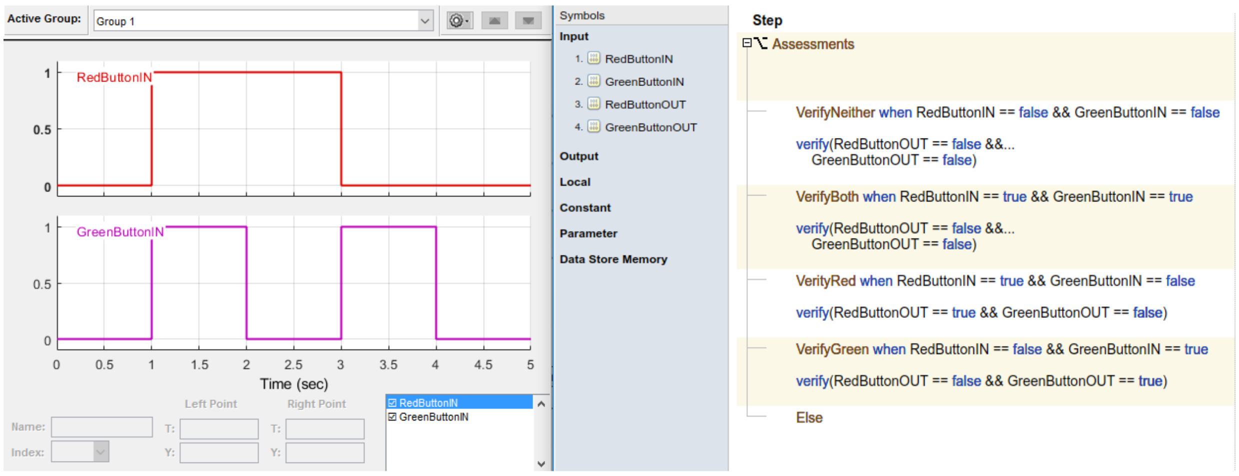
Activate verify Statements with Signal Conditions

If your model does not use a Test Sequence block source, or if Test Sequence steps do not correspond with conditions to verify, use unique signal conditions to activate `verify` statements. Place `verify` statements in a `When` decomposition sequence, and use conditional statements in the `When` conditions.

For example, this test harness uses a Signal Builder block input.



The Test Assessment block contains a `When` decomposition sequence. Each substep contains a `verify` statement. A unique signal condition activates each substep.



```
VerifyNeither when RedButtonIN == false && GreenButtonIN == false
verify(RedButtonOUT == false &&...
   GreenButtonOUT == false)
VerifyBoth when RedButtonIN == true && GreenButtonIN == true
verify(RedButtonOUT == false &&...
   GreenButtonOUT == false)
VerityRed when RedButtonIN == true && GreenButtonIN == false
verify(RedButtonOUT == true && GreenButtonOUT == false)
VerifyGreen when RedButtonIN == false && GreenButtonIN == true
verify(RedButtonOUT == false && GreenButtonOUT == true)
Else
```

Author verify Statements

`verify` statements evaluate logical expressions. You can label results in the Test Manager with optional arguments.

A `verify` statement returns a `pass`, `fail`, or `untested` result for each time step and for the overall simulation. A `fail` at any time step results in an overall `fail`. If there are no failing results, a `pass` at any time step results in an overall `pass`. Otherwise, the overall result is `untested`. Results appear in the **Verify Statements** section of the test results.

Syntax

`verify` statements use the syntax:

```
verify(expression)
verify(expression,errorMessage)
verify(expression,identifier,errorMessage)
```

where `expression`is a logical expression. Use additional arguments to define an `errorMessage` and a statement `identifier`. Error messages display in the diagnostic viewer. You can use error messages to display key values at the time the statement fails.

For example, if `verify` evaluates an expression containing variables `x` and `y`, you can display the values of `x` and `y` using the string:

```
'x and y values are %d, %d',x,y
```

An identifier labels the `verify` results in the Test Manager. The identifier uses a string of the form `'prefix:suffix'`. `prefix` and `suffix` are alphanumeric strings. For example:

```
'SimulinkTest:x_equals_y'
```

verify Statement Considerations

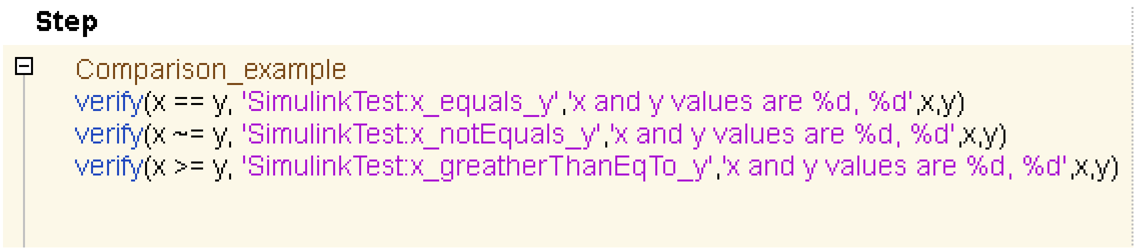
- `verify` is not supported in Test Sequence blocks that use continuous-time updating. Test Sequence block data can depend on factors such as the solver step time. Continuous-time updating can cause differences in when block data and `verify` statements update, which can lead to unexpected `verify` statement results.

  If your model uses continuous time and you use `verify` statements in a Test Sequence or Test Assessment block, consider explicitly setting a discrete block sample time.
- If you use parallel test execution to run your tests, then you cannot use the **Highlight in Model** button for `verify` results.

Example

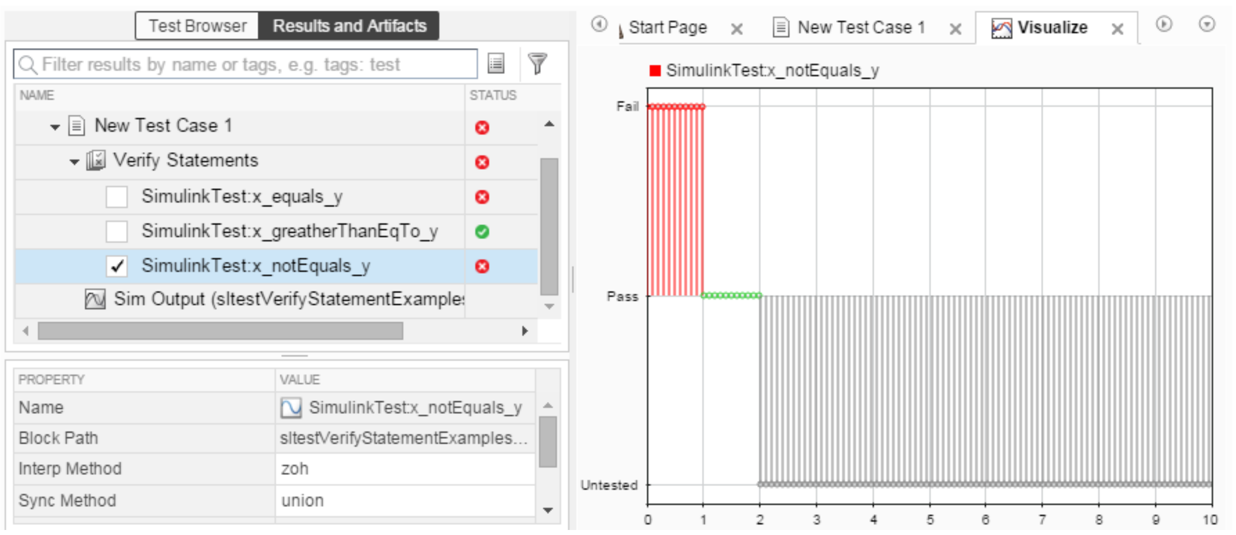
In this comparison of two values, the parent step uses `verify` statements to assess two local variables `x` and `y` during the simulation.

- `verify(x >= y)` passes overall because it is true for the entire test sequence.
- `verify(x == y)` and `verify(x ~= y)` fail because they fail in `step_1_2` and `step_1_1`, respectively.



```
Comparison_example
verify(x == y, 'SimulinkTest:x_equals_y','x and y values are %d, %d',x,y)
verify(x ~= y, 'SimulinkTest:x_notEquals_y','x and y values are %d, %d',x,y)
verify(x >= y, 'SimulinkTest:x_greatherThanEqTo_y','x and y values are %d, %d',x,y)
```

The Test Manager displays the results:


Test Browser
Results and Artifacts
Filter results by name or tags, e.g. tags: test
NAME
STATUS
New Test Case 1
Verify Statements
SimulinkTest:x_equals_y
SimulinkTest:x_greatherThanEqTo_y
SimulinkTest:x_notEquals_y
Sim Output (sltestVerifyStatementExample
PROPERTY
VALUE
Name
SimulinkTest:x_notEquals_y
Block Path
sltestVerifyStatementExamples...
Interp Method
zoh
Sync Method
union
Start Page
New Test Case 1
Visualize
SimulinkTest:x_notEquals_y
Fail
Pass
Untested
0
1
2
3
4
5
6
7
8
9
10

See Also

"Test Sequence Editor" on page 3-24 | Test Sequence | Test Assessment

Related Examples

- "Verify Multiple Conditions at a Time" on page 3-22
- "Requirements-Based Testing for Model Development"

Verify Multiple Conditions at a Time

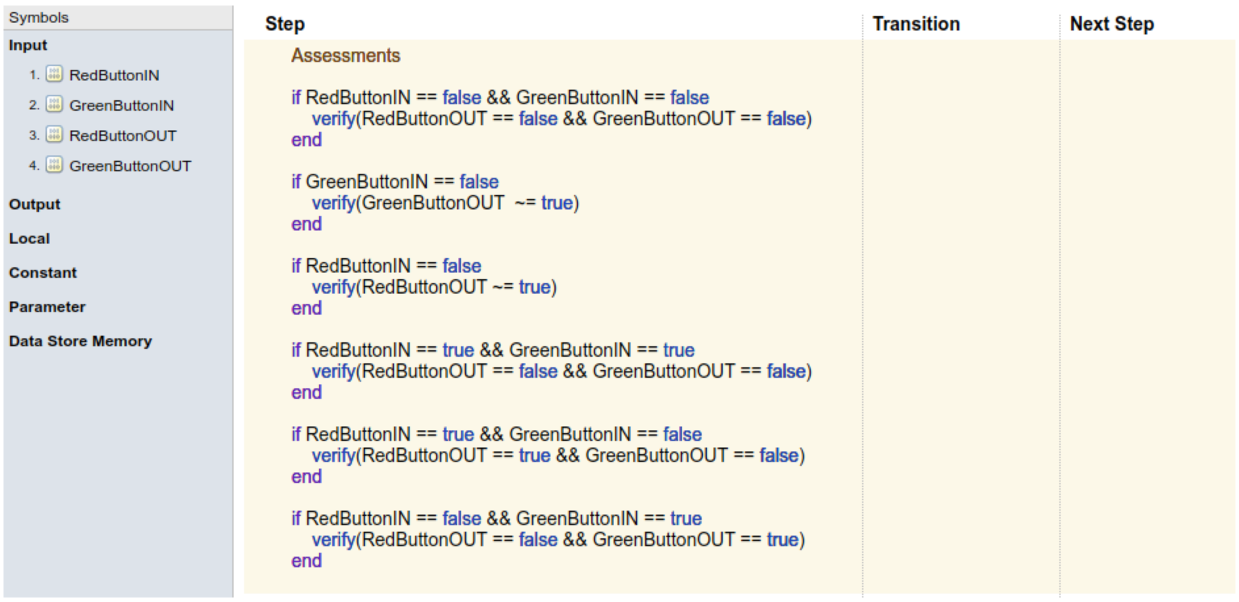
To verify multiple conditions in a single time step, include `verify` statements inside `if` statements, and include multiple `if` statements in a single test step.

For example, suppose you have a simple two-button utility function that operates as exclusive-or logic. More than one of the following conditions can be valid at the same time step.

Parallel Input Conditions and Expected Outputs

Condition	Expected Output
`RedButtonIN == false && GreenButtonIN == false`	`RedButtonOUT == false && GreenButtonOUT == false`
`GreenButtonIN == false`	`GreenButtonOUT ~= true`
`RedButtonIN == false`	`RedButtonOUT ~= true`
`RedButtonIN == true && GreenButtonIN == true`	`RedButtonOUT == false && GreenButtonOUT == false`
`RedButtonIN == true && GreenButtonIN == false`	`RedButtonOUT == true && GreenButtonOUT == false`
`RedButtonIN == false && GreenButtonIN == true`	`RedButtonOUT == false && GreenButtonOUT == true`

To assess these conditions, this Test Assessment block includes six `verify` statements in the first test step, contained in `if` statements. The test step is active during simulation, and the `if` statements are evaluated at each time step.



```
Assessments

if RedButtonIN == false && GreenButtonIN == false
    verify(RedButtonOUT == false && GreenButtonOUT == false)
end

if GreenButtonIN == false
    verify(GreenButtonOUT  ~= true)
end

if RedButtonIN == false
    verify(RedButtonOUT ~= true)
end

if RedButtonIN == true && GreenButtonIN == true
    verify(RedButtonOUT == false && GreenButtonOUT == false)
end

if RedButtonIN == true && GreenButtonIN == false
    verify(RedButtonOUT == true && GreenButtonOUT == false)
end

if RedButtonIN == false && GreenButtonIN == true
    verify(RedButtonOUT == false && GreenButtonOUT == true)
end
```

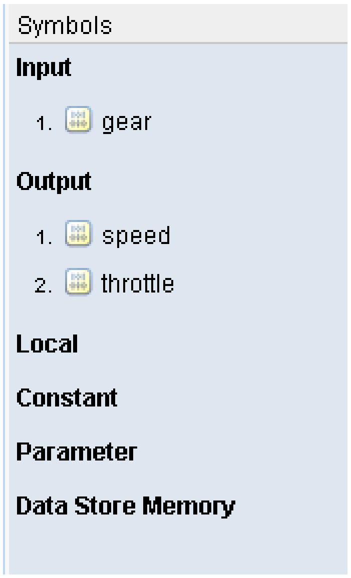
See Also

Test Sequence Editor

Input, Output, and Data Management

Manage inputs, outputs, and data objects using the **Symbols** sidebar of the Test Sequence Editor. Click the symbols sidebar button on the toolbar to show or hide the sidebar. To add a symbol, hover over the symbol type and click **Add**. To edit or delete a data symbol, hover over the data symbol and click **Edit** or **Delete**.

If you add a symbol to the test sequence block, you can access that symbol from test steps at any hierarchy level. For information on using messages, see "Actions and Transitions" on page 3-38.



Symbols

Input

1. gear

Output

1. speed
2. throttle

Local

Constant

Parameter

Data Store Memory

Symbol Type	Description	Procedure for Adding
Input	Inputs can be data or messages.	Click **Add** in the sidebar and enter the input name.
Output	Outputs can be data, messages, or function calls.	Click **Add** in the sidebar and enter the output name.

Symbol Type	Description	Procedure for Adding
Local	Local data entries are available inside the test sequence block in which they are defined.	Add a local variable in the sidebar and initialize the local variable in the first test step.
Constant	Constants are read-only data entries available inside the test sequence block in which they are defined.	Add a constant in the sidebar. Click **Edit** and enter the constant value in the dialog box, in the **Initial Value** field.
Parameter	Parameters are data available inside and outside the Test Sequence block.	Using the Model Explorer, add a parameter in the workspace of the model containing the Test Sequence block. Then add the parameter name to the **Parameter** symbols.
Data Store Memory	Data Store Memory entries are available inside and outside the Test Sequence block.	Using the Model Explorer, add a Simulink.signal entry in the workspace of the model containing the Test Sequence block. Alternatively, add a Data Store Memory block to the model. Then add the data store memory name to the **Data Store Memory** symbols.

Find and Replace

You can find and replace text in Test Sequence actions, transitions, and descriptions by using the **Find & Replace** tool in the Test Sequence Editor:

1

To open the **Find & Replace** tool, click the icon in the toolbar.

2 In the **Find what:** box, enter the text you want to locate.
3 In the **Replace with:** with box, enter the updated text.
4 To locate the text, click **Find Next** or **Find Previous**.
5 Click **Replace** to replace the old text with the updated text.

When running a search, the **Find & Replace** tool searches descriptions only if the description column is open.

Add and Delete Test Steps

To add a test step, right-click a step. Select **Add step before** or **Add step after**. Select **Add sub-step** to create a test step in a lower hierarchy level.

To delete a test step, right-click the step. Select **Delete step**. If the sequence contains only one test step, you cannot delete it. You can delete the contents by selecting **Erase last step content**.

Automatic Syntax Correction

The Test Sequence Editor changes the following syntax automatically:

- **Duplicate test step names**. For example, if `step_1` exists, and you change another step name to `step_1`, the step name you change automatically changes to `step_2`.
- **Increment and decrement operations** to use MATLAB as the action language, such as `a++` and `a--`. For example, `a++` is changed to `a=a+1`.
- **Assignment operations** to use MATLAB as the action language, such as `a+=expr`, `a–=expr`, `a*=expr`, and `a/=expr`. For example, `a+=b` is changed to `a=a+b`.
- **Evaluation operations** to use MATLAB as the action language, such as `a!=expr` and `!a`. For example, `a!=b` is changed to `a~=b`.
- **Explicit casts for literal constant assignments**. For example, if `y` is defined as type `single`, then `y=1` is changed to `y=single(1)`.

Copy Test Steps

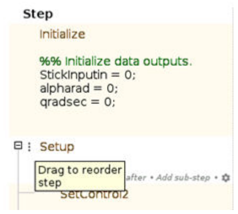
You can cut or copy test steps, and paste them before or after another step. You can also paste them in a hierarchy below another step. Right-click the test step and select **Cut step** or **Copy step**. To paste, right-click another test step and select **Paste step > Paste**

before step or **Paste step** > **Paste after step**. To paste in a lower hierarchy, select **Paste step** > **Paste sub-step**.

You can also use **Ctrl+X**, **Ctrl+C**, and **Ctrl+V** shortcuts.

Reorder Test Steps and Transitions

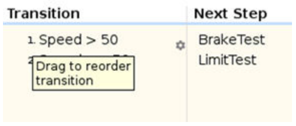
You can reorder test steps from the editor. Hover over the icon to the left of the step name. Click and drag the icon to reorder the test step. Test steps can be reordered within the same hierarchy level. Substeps are also moved with the test step.


Step
Initialize

```
%% Initialize data outputs.
StickInputin = 0;
alpharad = 0;
qradsec = 0;
```

Setup

You can reorder step transitions within the same test step. Hover over the transition number. Click and drag the number to reorder the transition. The corresponding next step is maintained.



Transition	Next Step
1. Speed > 50	BrakeTest
	LimitTest

Change Test Step Hierarchy

Change test step hierarchy level by indenting or outdenting the test step. Right-click the test step, and select **Indent step** to move it to a lower level, or **Outdent step** to move it to a higher level.

- Moving to a lower hierarchy level (indenting) requires a preceding step at the same hierarchy level. You cannot indent the first test step in a sequence or the first step in a hierarchy group.
- Only the last step in a hierarchy group can be moved to a higher level.

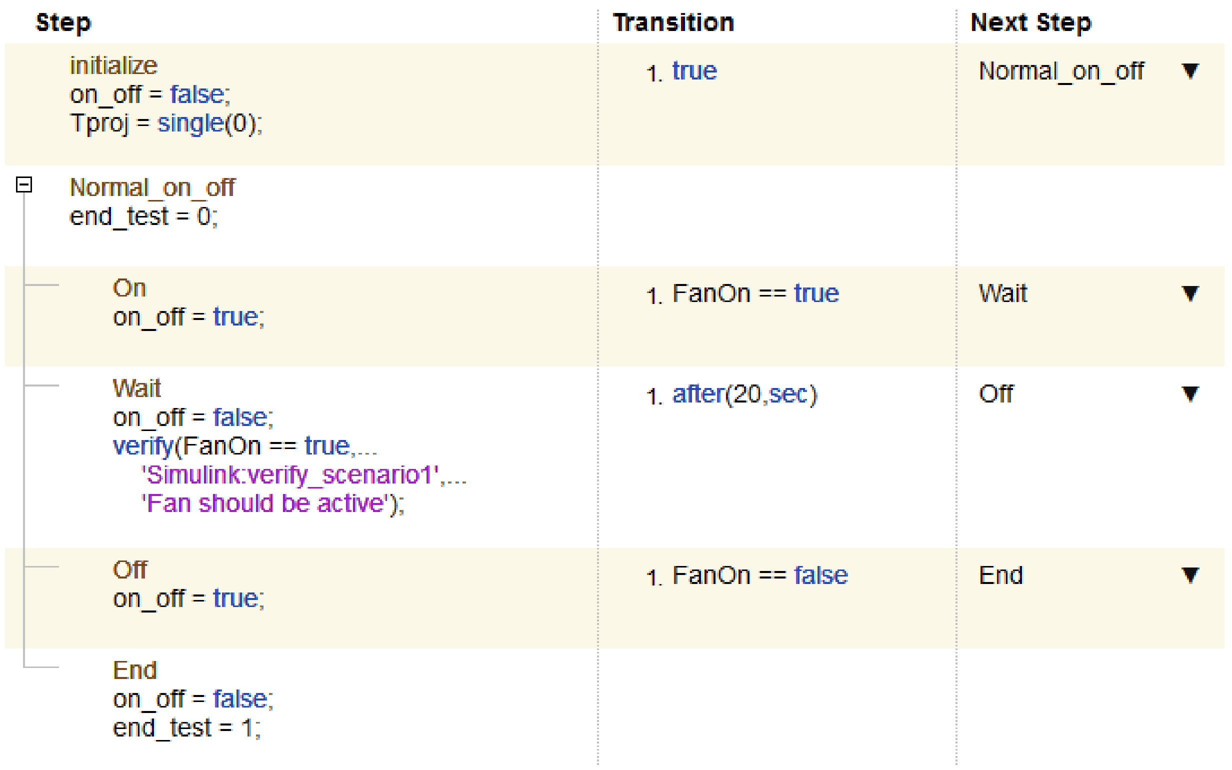
Standard Test Step Sequences

A standard sequence progresses according to transition conditions and next steps. The default step is the first test step listed in the sequence. To create a standard sequence:

- Add new steps to the sequence.
- Define outputs and assessments in the **Step** cell. For example, this code sets `on_off` to `false` and verifies that the `FanOn` signal is `true`.

  ```
  on_off = false;
  verify(FanOn == true);
  ```
- For each step that requires a transition, hover over the **Transition** cell and click **Add transition**. Define the step exit conditions in the transition. For example, this code transitions to another step after the current step has been active for `20` seconds.

  ```
  after(20,sec)
  ```
- Select the next test step from the drop-down list in the **Next Step** cell.



Step	Transition	Next Step

initialize
on_off = false;
Tproj = single(0);
	1. true	Normal_on_off ▼

⊟ Normal_on_off end_test = 0;		
On on_off = true;	1. FanOn == true	Wait ▼
Wait on_off = false; verify(FanOn == true,... 'Simulink:verify_scenario1',... 'Fan should be active');	1. after(20,sec)	Off ▼
Off on_off = true;	1. FanOn == false	End ▼

End
on_off = false;
end_test = 1;		

When Decomposition Sequences

A `When` decomposition sequence contains a set of two or more substeps. Conceptually, a `When` decomposition sequence uses logic similar to `if-elseif-else`. You define a logical condition preceded by `When` for each substep, except for the final substep. The logical conditions determine which step is active at a given time.

The final substep is not assigned a `When` condition. Similar to `else`, the final step is active if a simulation condition does not match a `When` condition in the sequence. Each substep in the `When` decomposition can contain actions. At each time step, the `when` statements evaluate from first to last, and the first step with a valid condition activates.

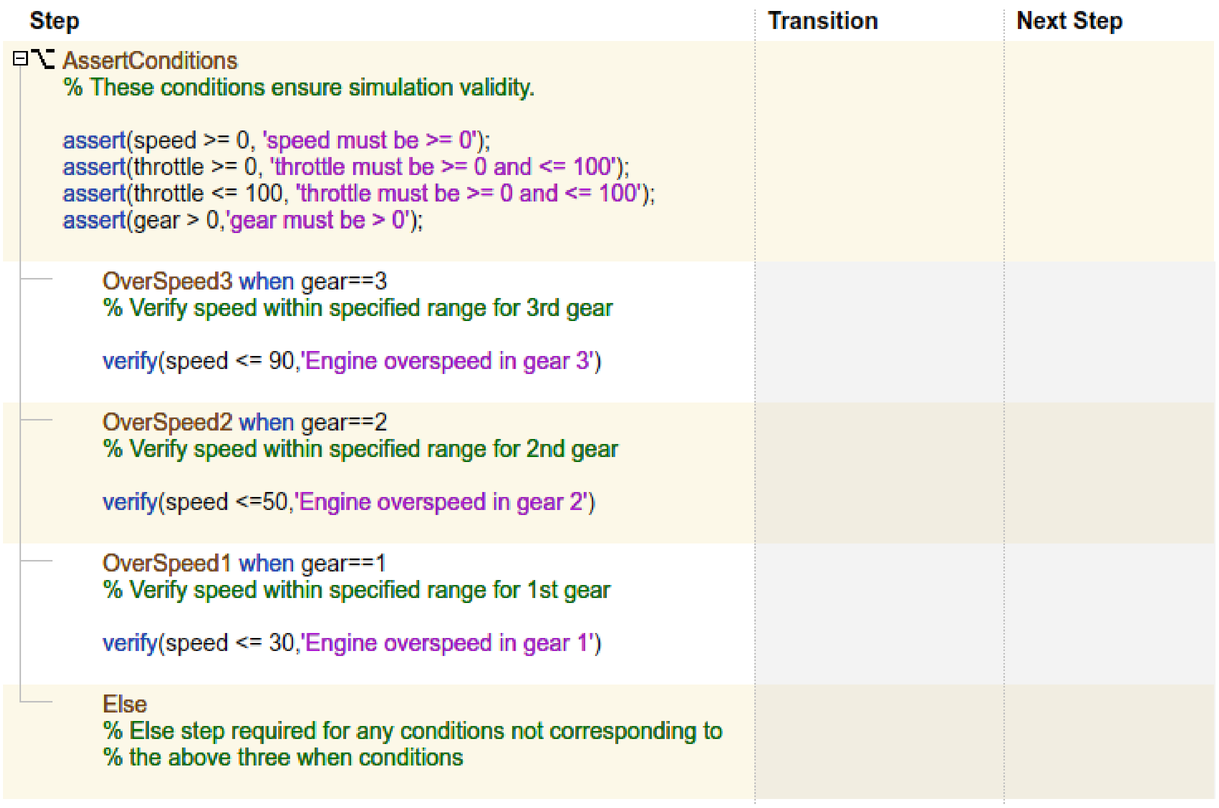
To create a `When` decomposition sequence, right-click a test step and select **When decomposition**. The step displays the icon ⌥. The Test Assessment block contains an empty `When` decomposition by default:

Add substeps to include more conditions in the sequence. Precede each condition with the `when` operator. For example, to create a step named `OverSpeed2` that activates when `gear` is equal to `2`, enter:

```
OverSpeed2 when gear == 2
```

This example `When` decomposition sequence contains:

- Assertions in the parent step `AssertConditions`. If one of the `assert` statements become `true`, simulation stops.
- Three `OverSpeed` steps with `When` conditions. Each step activates with a value of `gear`. A `verify` statement in each step verifies that `speed` remains below a limit.
- The `Else` step, which does not contain actions in this example.



Step	Transition	Next Step

AssertConditions
% These conditions ensure simulation validity.
 	
assert(speed >= 0, 'speed must be >= 0');
assert(throttle >= 0, 'throttle must be >= 0 and <= 100');
assert(throttle <= 100, 'throttle must be >= 0 and <= 100');
assert(gear > 0,'gear must be > 0');		

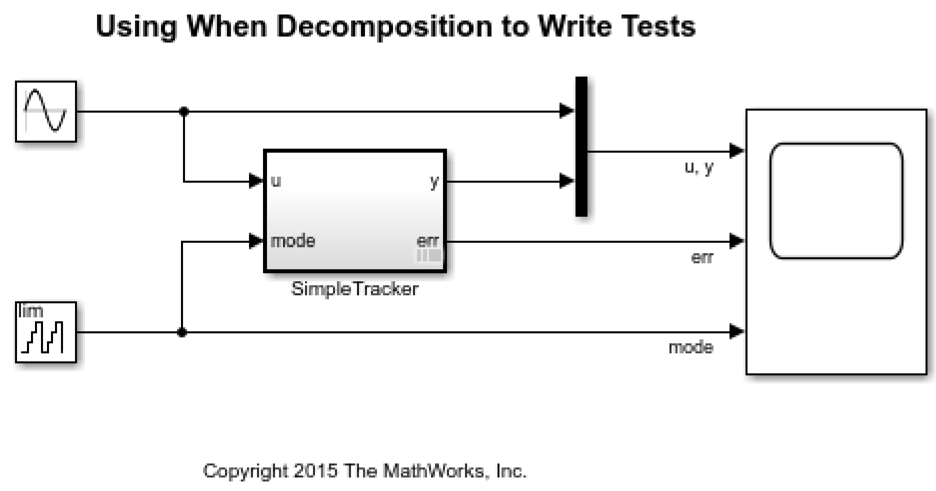
OverSpeed3 when gear==3 % Verify speed within specified range for 3rd gear verify(speed <= 90,'Engine overspeed in gear 3')		
OverSpeed2 when gear==2 % Verify speed within specified range for 2nd gear verify(speed <=50,'Engine overspeed in gear 2')		
OverSpeed1 when gear==1 % Verify speed within specified range for 1st gear verify(speed <= 30,'Engine overspeed in gear 1')		
Else % Else step required for any conditions not corresponding to % the above three when conditions		

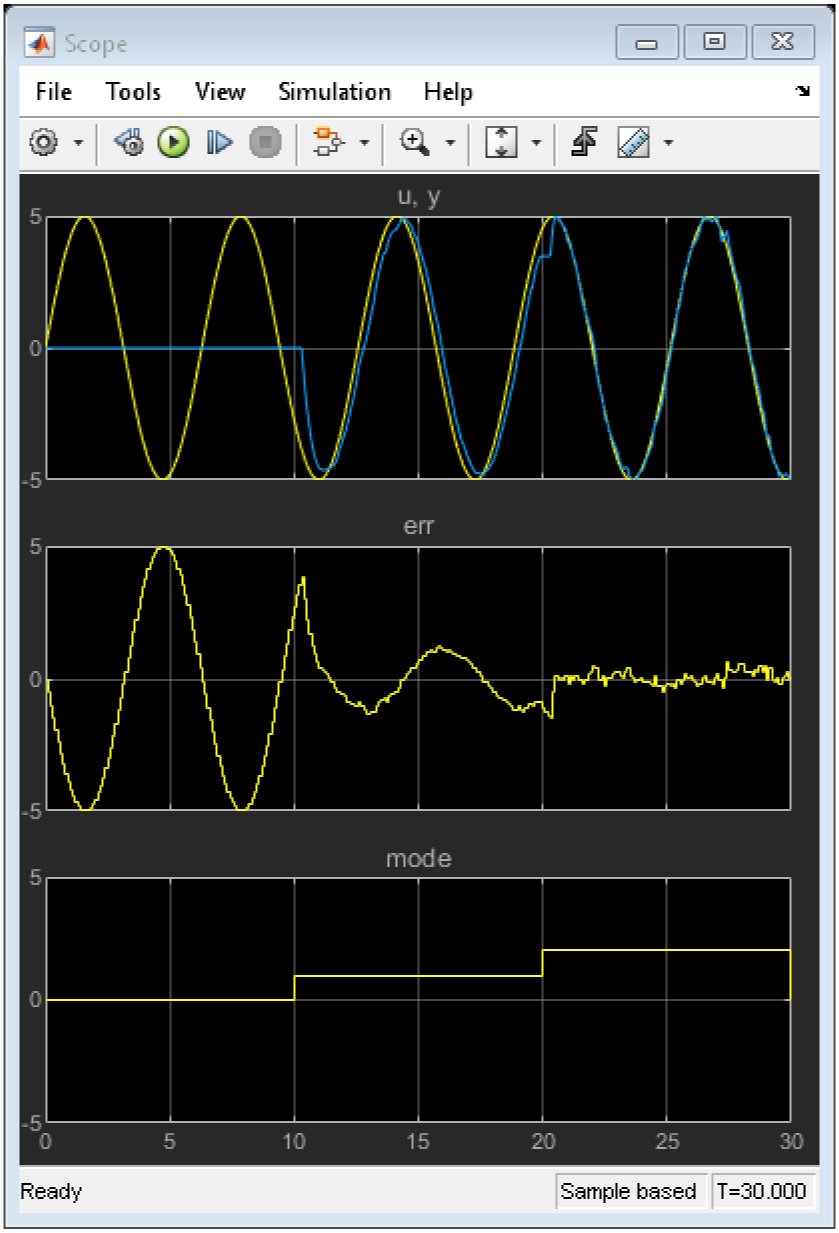
Using When Decomposition to Write Tests

Assess a model using a When decomposition sequence.

This example shows how to use When decomposition in a Test Sequence block to author assessments in a test harness. The example model implements a simple signal tracker that operates in three modes: off (0), slow (1) and quick (2). Simulate the model and observe the output and error of the signal tracker.

```
mdl = 'sltestTestSequenceWhenExample';
open_system(mdl);
open_system([mdl '/Scope']);
sim(mdl);
```

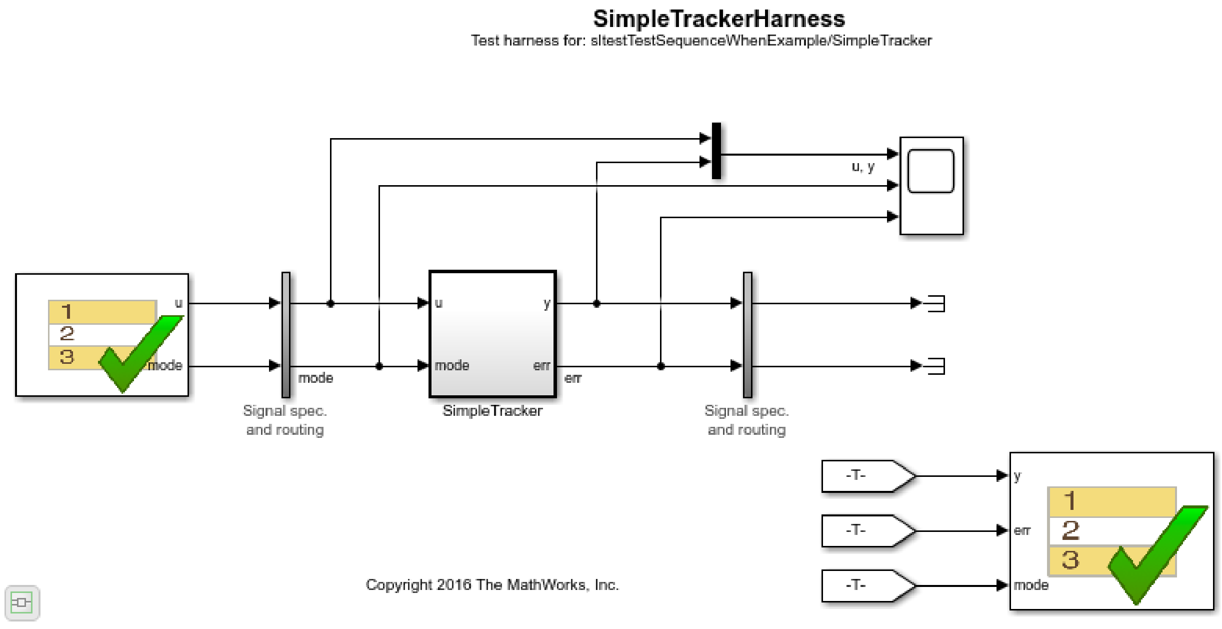

Using When Decomposition to Write Tests
u
y
mode
err
SimpleTracker
lim
u, y
err
mode
Copyright 2015 The MathWorks, Inc.

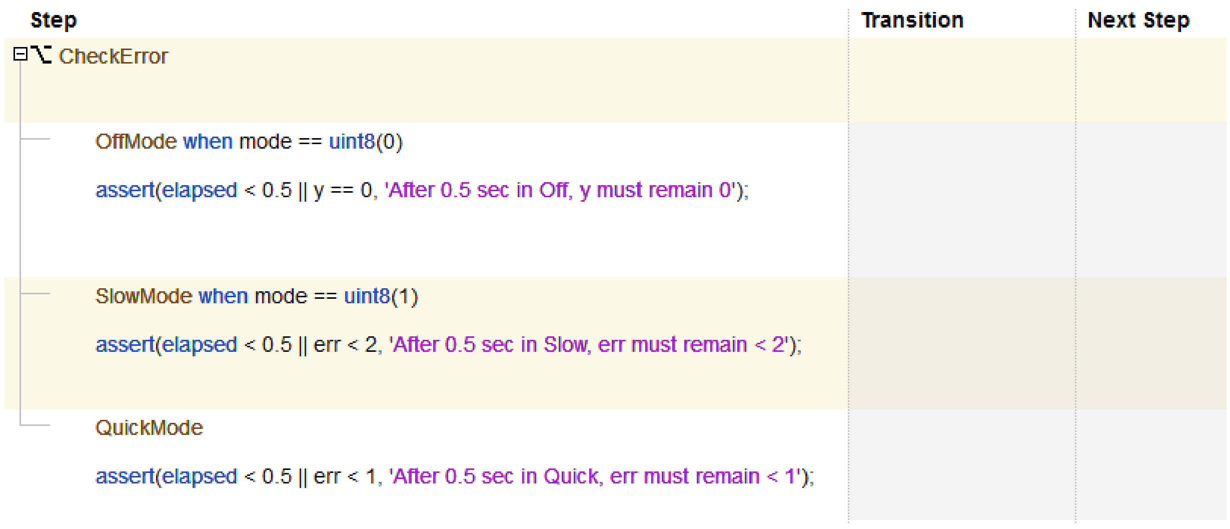

Scope
File
Tools
View
Simulation
Help
u, y
err
mode
5
0
-5
0
5
10
15
20
25
30
Ready
Sample based
T=30.000

Open the test harness attached to the `SimpleTracker` subsystem, and open the Test Sequence block named `Test Assessment` that assesses the behavior of `SimpleTracker`.

The Test Sequence block uses When decomposition to determine the appropriate assertions to run depending on the `SimpleTracker` mode. The `CheckError` step is a When decomposition step, and it has three substeps, `OffMode`, `SlowMode`, and `QuickMode` that are active when mode is **0**, **1** or **otherwise**, respectively.

```
open_system(mdl);
sltest.harness.open([mdl '/SimpleTracker'],'SimpleTrackerHarness');
open_system('SimpleTrackerHarness/Test Assessment');
```

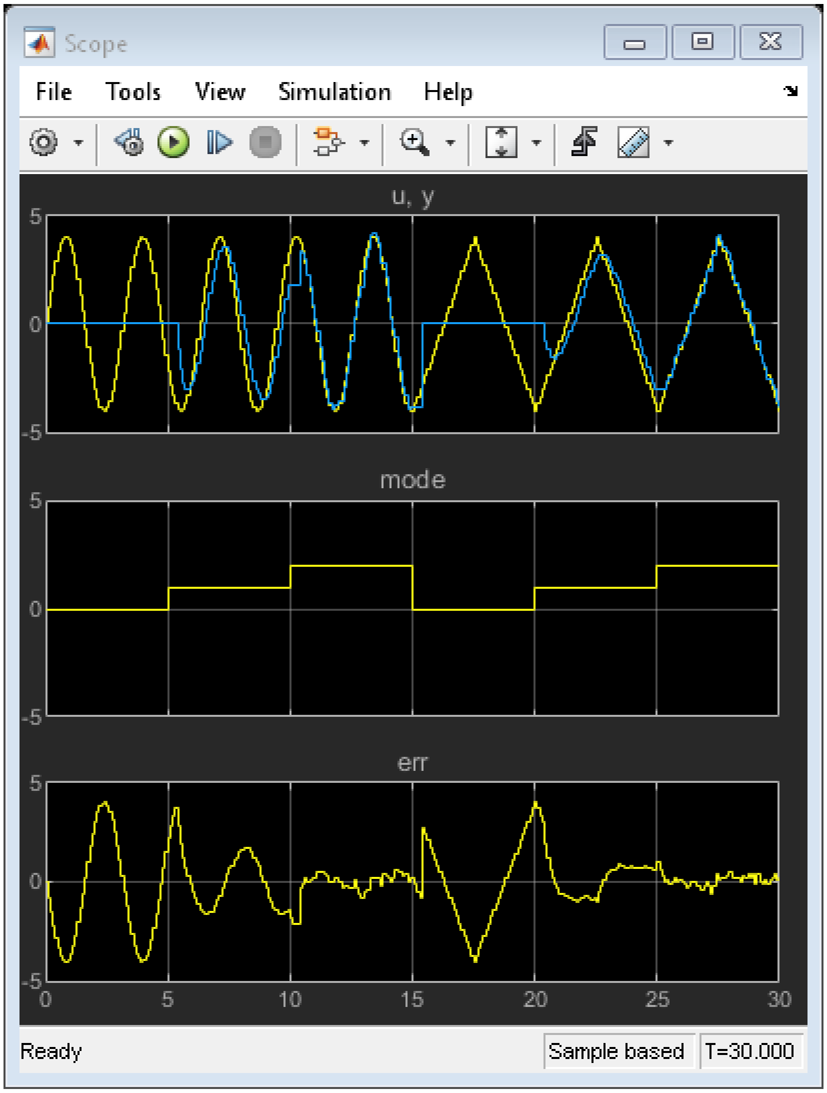




Step	Transition	Next Step
⊟ CheckError		
OffMode when mode == uint8(0) assert(elapsed < 0.5 || y == 0, 'After 0.5 sec in Off, y must remain 0');		
SlowMode when mode == uint8(1) assert(elapsed < 0.5 || err < 2, 'After 0.5 sec in Slow, err must remain < 2');		
QuickMode assert(elapsed < 0.5 || err < 1, 'After 0.5 sec in Quick, err must remain < 1');		

Simulate the test harness to run the assessments.

```
open_system('SimpleTrackerHarness/Scope');
sim('SimpleTrackerHarness');
```



Close the test harness and main model.

```
close_system(mdl, 0);
```

```
clear mdl;
```

See Also

Test Sequence | Test Assessment | “Test Sequence and Assessment Syntax” on page 3-60

Related Examples

- “Programmatically Create a Test Sequence” on page 3-55

Actions and Transitions

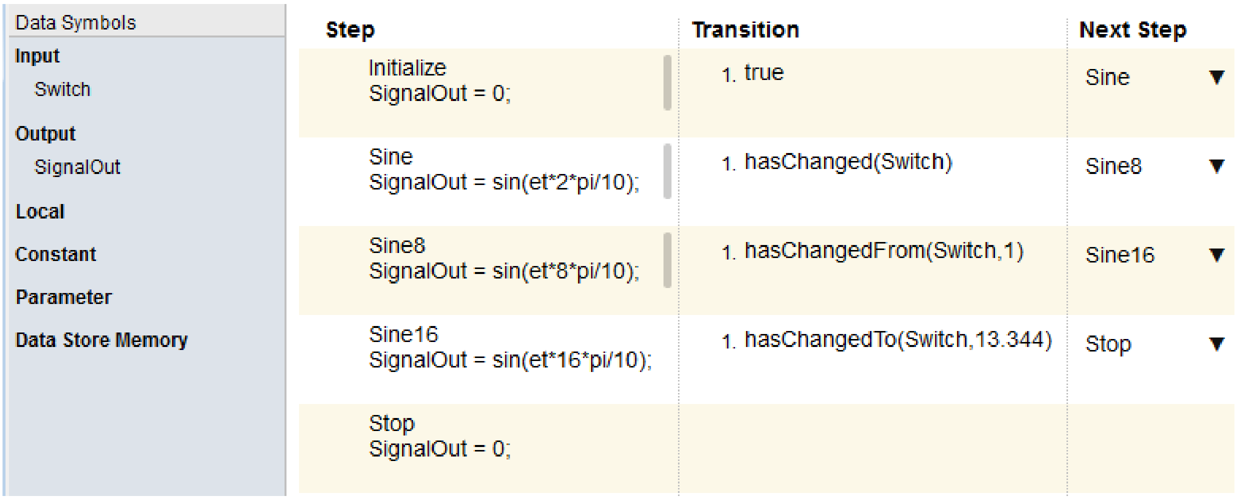
In this section...

Transition Between Steps Using Temporal or Signal Conditions

The Test Sequence block uses MATLAB as the action language. You can transition between test steps by evaluating the component under test. You can use conditional logic, temporal operators, and event operators.

Consider a simple test sequence that outputs a sine wave at three frequencies. The test sequence transitions between steps:

- From `Initialize` to `Sine` when `Switch` changes
- From `Sine` to `Sine8` when `Switch` changes from the value `1`
- From `Sine8` to `Sine16` when `Switch` changes to the value `13.344`



Step	Transition	Next Step
Initialize SignalOut = 0;	1. true	Sine
Sine SignalOut = sin(et*2*pi/10);	1. hasChanged(Switch)	Sine8
Sine8 SignalOut = sin(et*8*pi/10);	1. hasChangedFrom(Switch,1)	Sine16
Sine16 SignalOut = sin(et*16*pi/10);	1. hasChangedTo(Switch,13.344)	Stop
Stop SignalOut = 0;		

Temporal Operators

To create an expression that evaluates the simulation time, use temporal operators. Variables used in signal conditions must be inputs, parameters, or constants in the Test Sequence block.

Operator	Syntax	Description	Example
`et`	`et(TimeUnits)`	The elapsed time of the test step in `TimeUnits`. Omitting `TimeUnits` returns the value in seconds.	The elapsed time of the test sequence step in milliseconds: `et(msec)`
`t`	`t(TimeUnits)`	The elapsed time of the simulation in `TimeUnits`. Omitting `TimeUnits` returns the value in seconds.	The elapsed time of the simulation in microseconds: `t(usec)`
`after`	`after(n, TimeUnits)`	Returns `true` if `n` specified units of time in `TimeUnits` elapse since the beginning of the current test step.	After 4 seconds: `after(4,sec)`
`before`	`before(n, TimeUnits)`	Returns `true` until `n` specified units of time in `TimeUnits` elapse, beginning with the current test step.	Before 4 seconds: `before(4,sec)`
`duration`	`ElapsedTime = duration (Condition, TimeUnits)`	Returns `ElapsedTime` in `TimeUnits` for which `Condition` has been `true`. `ElapsedTime` is reset when the test step is re-entered or when `Condition` is no longer `true`.	Return `true` if the time in milliseconds since `Phi > 1` is greater than 550: `duration(Phi>1,msec) > 550`

Syntax in the table uses these arguments:

`TimeUnits`

The units of time

Value: `sec|msec|usec`

Examples:

```
msec
```

`Condition`

Logical expression triggering the operator. Variables used in `duration` can be inputs, parameters, or constants, with at most one local or output data.

Examples:

```
u > 0
x <= 1.56
```

Transition Operators

To create expressions that evaluate signal events, use transition operators. Common transition operators include:

Operator	Syntax	Description	Example
`hasChanged`	`hasChanged(u)`	Returns `true` if `u` changes in value since the beginning of the test step, otherwise returns `false`. `u` must be an input data symbol.	Transition when `h` changes: `hasChanged(h)`

Operator	Syntax	Description	Example
`hasChangedFrom`	`hasChangedFrom(u,A)`	Returns true if `u` changes from the value `A`, otherwise returns false. `u` must be an input data symbol.	Transition when `h` changes from `1`: `hasChangedFrom(h,1)`
`hasChangedTo`	`hasChangedTo(u,B)`	Returns true if `u` changes to the value `B`, otherwise returns false. `u` must be an input data symbol.	Transition when `h` changes to `0`: `hasChangedTo(h,0)`

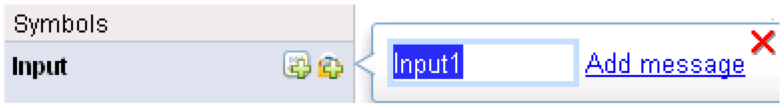
Use Messages in Test Sequences

Messages carry data between Test Sequence blocks and other blocks such as Stateflow® charts. Messages can be used to model asynchronous events. A message is queued until you evaluate it, which removes it from the queue. You can use messages and message data inside a test sequence. The message remains valid until you forward it, or the time step ends. For more information, see “Messages” (Stateflow) in the Stateflow® documentation.

Receive Messages and Access Message Data

If your Test Sequence block has a message input, you can use queued messages in test sequence actions or transitions. Use the `receive` command before accessing message data or forwarding a message.

To create a message input, hover over **Input** in the **Symbols** sidebar, click the add message icon, and enter the message name.

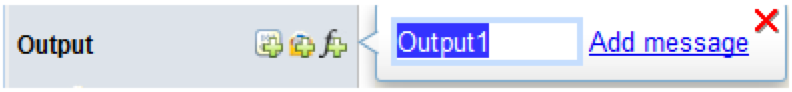


`receive(M)` determines whether a message is present in the input queue `M`, and removes the message from the queue. `receive(M)` returns `true` if a message is in the queue, and false if not. Once the message is received, you can access the message data using the dot notation, `M.data`, or forward the message. The message is valid until it is forwarded or the current time step ends.

The order of message removal depends on the queue type. Set the queue type using the message properties dialog box. In the **Symbols** sidebar, click the edit icon next to the message input, and select the **Queue type**.

Send Messages

To send a message, create a message output and use the `send` command. To create a message output, hover over **Output** in the **Symbols** sidebar, click the add message icon, and enter the message name.



You can assign data to the message using the dot notation `M.data`, where M is the message output of the Test Sequence block. `send(M)` sends the message.

Forward Messages

You can forward a message from an input message queue to an output port. To forward a message:

1. Receive the message from the input queue using `receive`.
2. Forward the message using the command `forward(M,M_out)` where `M` is the message input queue and `M_out` is the message output.

Compare Test Sequences Using Data and Messages

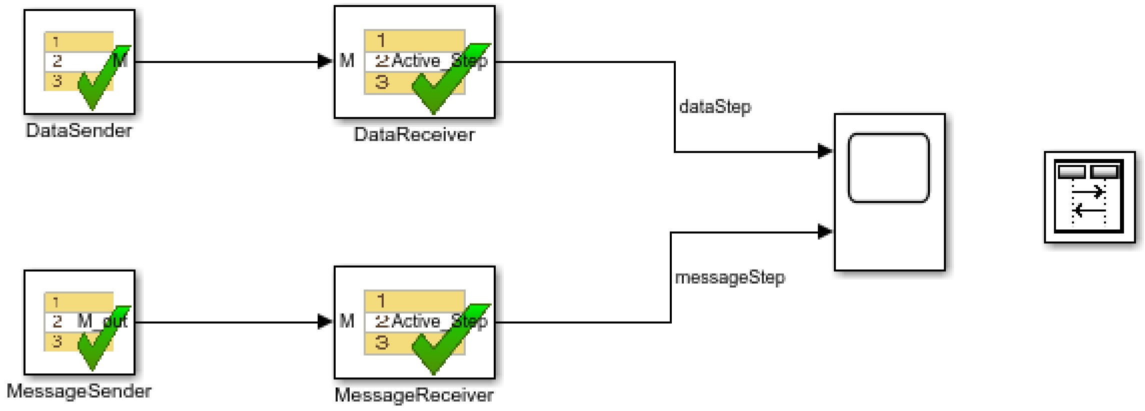
This example demonstrates message inputs and outputs, sending, and receiving a message. The model compares two pairs of test sequences. Each pair is comprised of a sending and receiving test sequence block. The first pair sends and receives data, and the second sends and receives a message.

Set the following path and model name variables.

```
filePath = fullfile(matlabroot,'examples','simulinktest');
model = 'sltest_testsequence_data_vs_message';
```

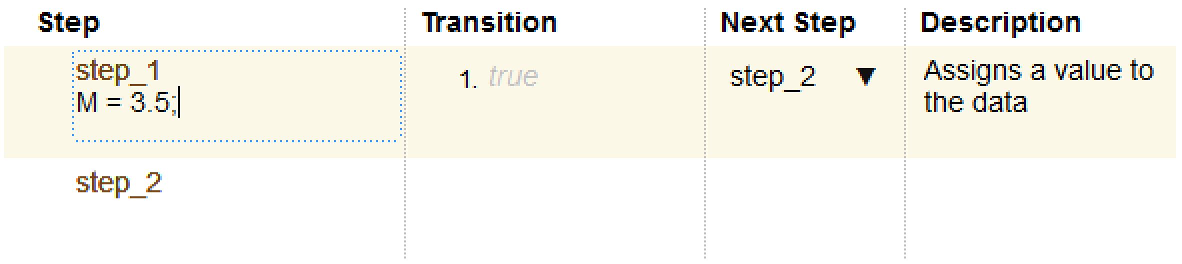
Open the model.

```
open_system(fullfile(filePath,model))
```



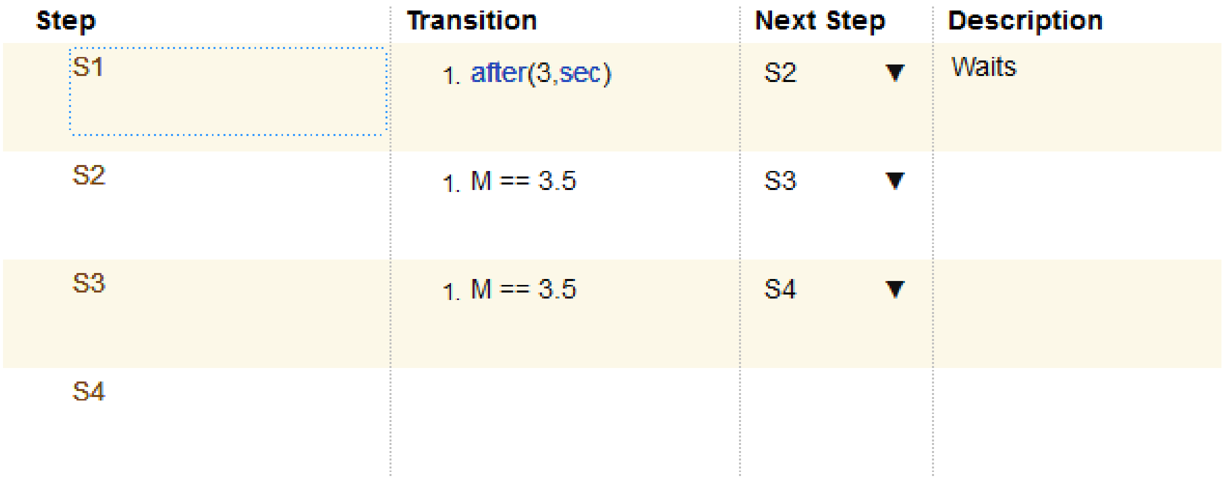
Test Sequences Using Data

The DataSender block assigns a value to a data output M.



Step	Transition	Next Step	Description
step_1 M = 3.5;	1. *true*	step_2 ▼	Assigns a value to the data
step_2			

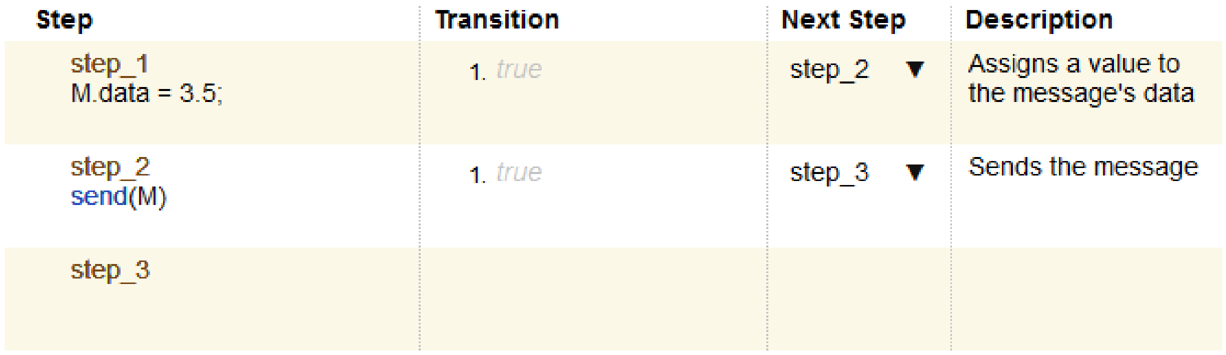
The DataReceiver block waits 3 seconds, then transitions to step S2. Step S2 transitions to step S3 using a condition comparing M to the expected value, and does the same for S3 to S4.



Step	Transition	Next Step	Description
S1	1. after(3,sec)	S2 ▼	Waits
S2	1. M == 3.5	S3 ▼	
S3	1. M == 3.5	S4 ▼	
S4			

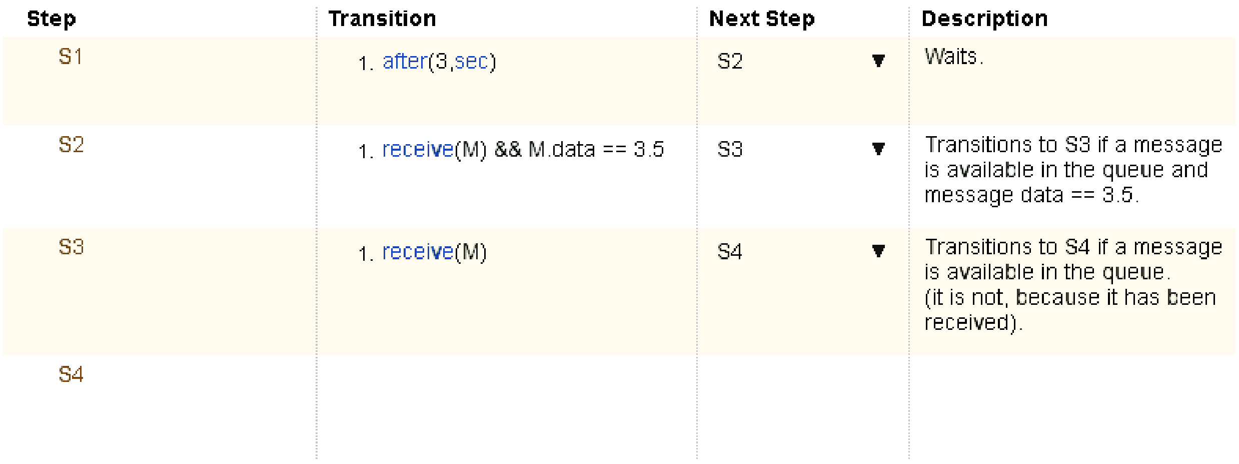
Test Sequences Using Messages

The MessageSender block assigns a value to the message data of a message output `M_out`, then sends the message to the MessageReceiver block.



Step	Transition	Next Step	Description
step_1 M.data = 3.5;	1. true	step_2 ▼	Assigns a value to the message's data
step_2 send(M)	1. true	step_3 ▼	Sends the message
step_3			

The MessageReceiver block waits 3 seconds, then transitions to step S2. Step S2's transition evaluates the queue `M` with `receive(M)`, removing the message from the queue. `receive(M)` returns `true` since the message is present. `M.data == 3.5` compares the message data to the expected value. The statement is true, and the sequence transitions to step S3.

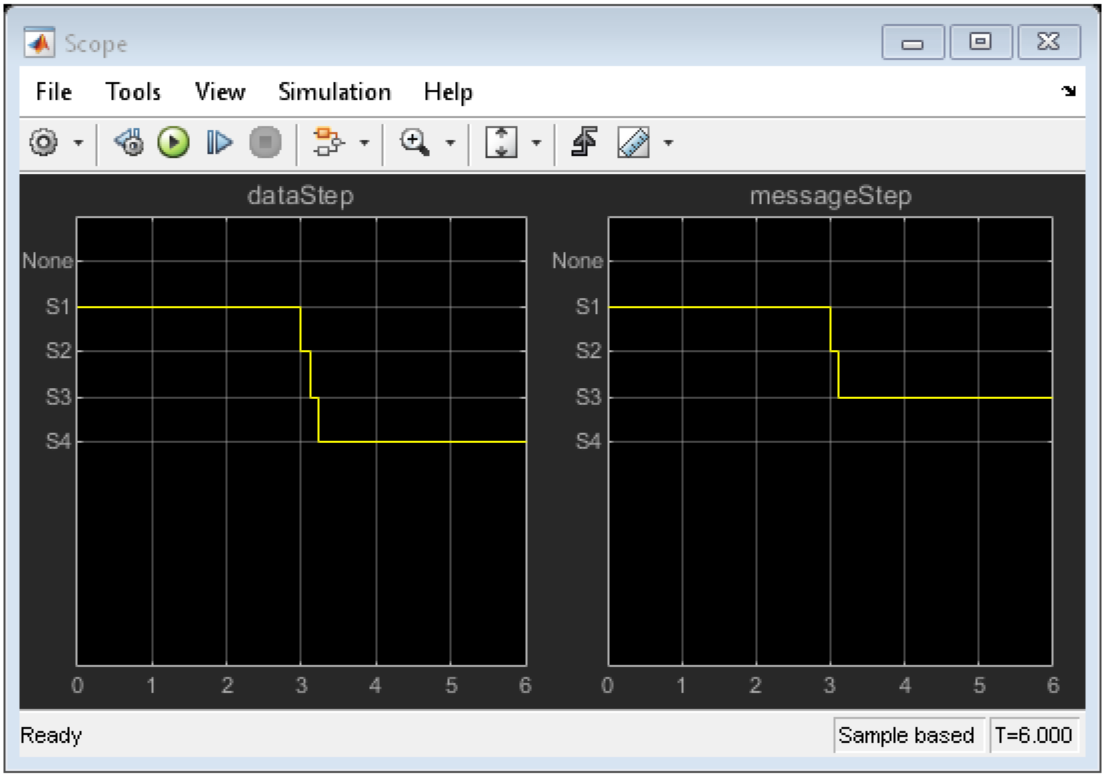


Step	Transition	Next Step	Description
S1	1. after(3,sec)	S2 ▼	Waits.
S2	1. receive(M) && M.data == 3.5	S3 ▼	Transitions to S3 if a message is available in the queue and message data == 3.5.
S3	1. receive(M)	S4 ▼	Transitions to S4 if a message is available in the queue. (it is not, because it has been received).
S4			

When step S3's transition condition evaluates, no messages are present in the queue. Therefore, S3 does not transition to S4.

Run the test and observe the output comparing the different behaviors of the test sequence pairs.

```
open_system([model '/Scope'])
sim(model)
```



```
close_system(model,0)
clear(model, filePath)
```

See Also

“Test Sequence and Assessment Syntax” on page 3-60 | Test Sequence

Related Examples

- “Assess Model Simulation Using verify Statements” on page 3-15
- “Signal Generation Functions” on page 3-47

Signal Generation Functions

In this section...

In the Test Sequence block, you can generate signals for testing.

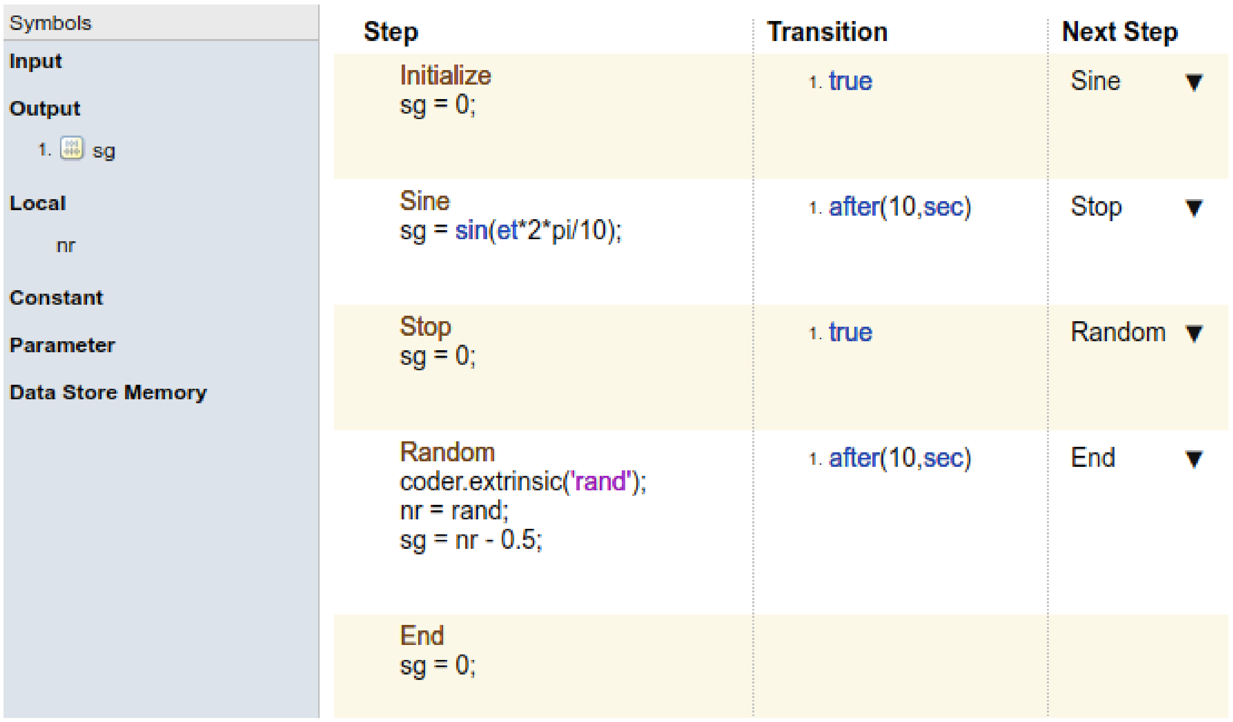
1. Define an output data symbol in the **Data Symbols** pane.
2. Use the output name with a signal generation function in the test step action.

You can call external functions from the Test Sequence block. Define a function in a script on the MATLAB path, and call the function in the test sequence.

Sinusoidal and Random Number Functions in Test Sequences

This example shows how to produce a sine and a random number test signal in a Test Sequence block.

The step `Sine` outputs a sine wave with a period of 10 seconds, specified by the argument `et*2*pi/10`. The step `Random` outputs a random number in the interval `-0.5 to 0.5`.

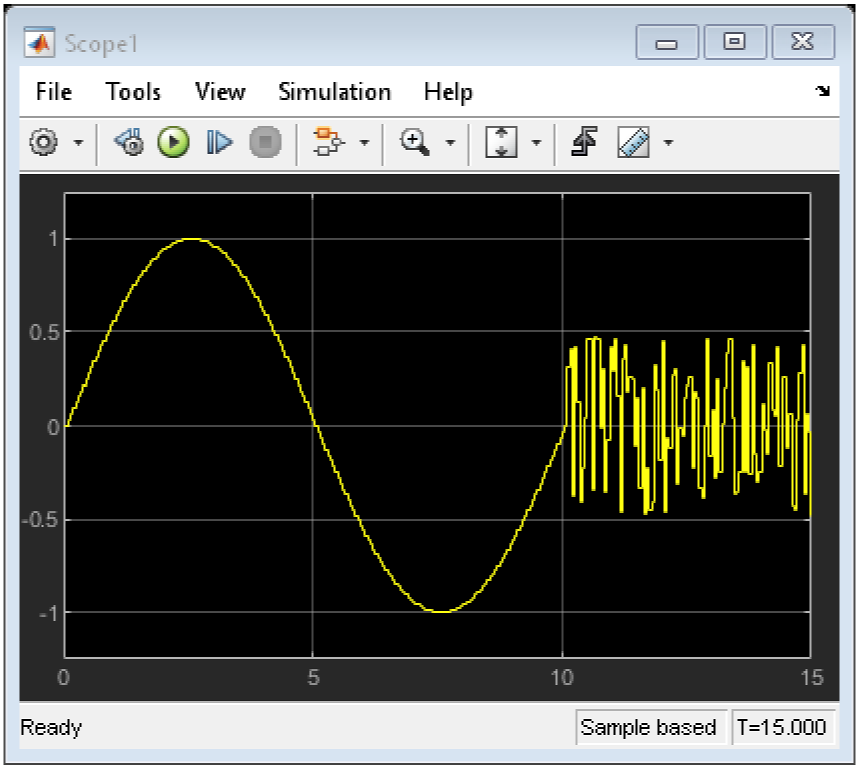


Step	Transition	Next Step
Initialize sg = 0;	1. true	Sine ▼
Sine sg = sin(et*2*pi/10);	1. after(10,sec)	Stop ▼
Stop sg = 0;	1. true	Random ▼

Random
coder.extrinsic('rand');
nr = rand;
sg = nr - 0.5;
	1. after(10,sec)	End ▼

End sg = 0;		

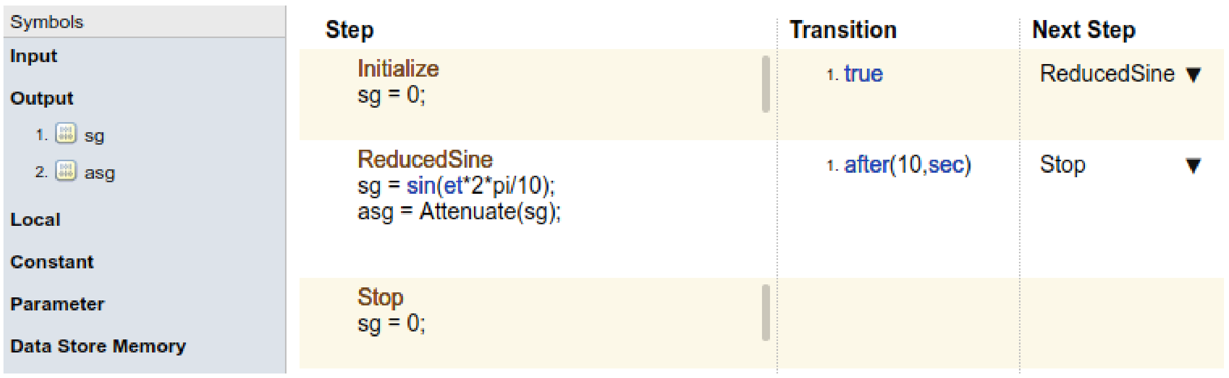
The test sequence produces signal `sg`.



Using an External Function from a Test Sequence Block

This example shows how to call an externally-defined function from the Test Sequence block. Define a function in a script on the MATLAB® path, and call the function from the test sequence.

In this example, the step `ReducedSine` reduces the signal `sg` using the function `Attenuate`.

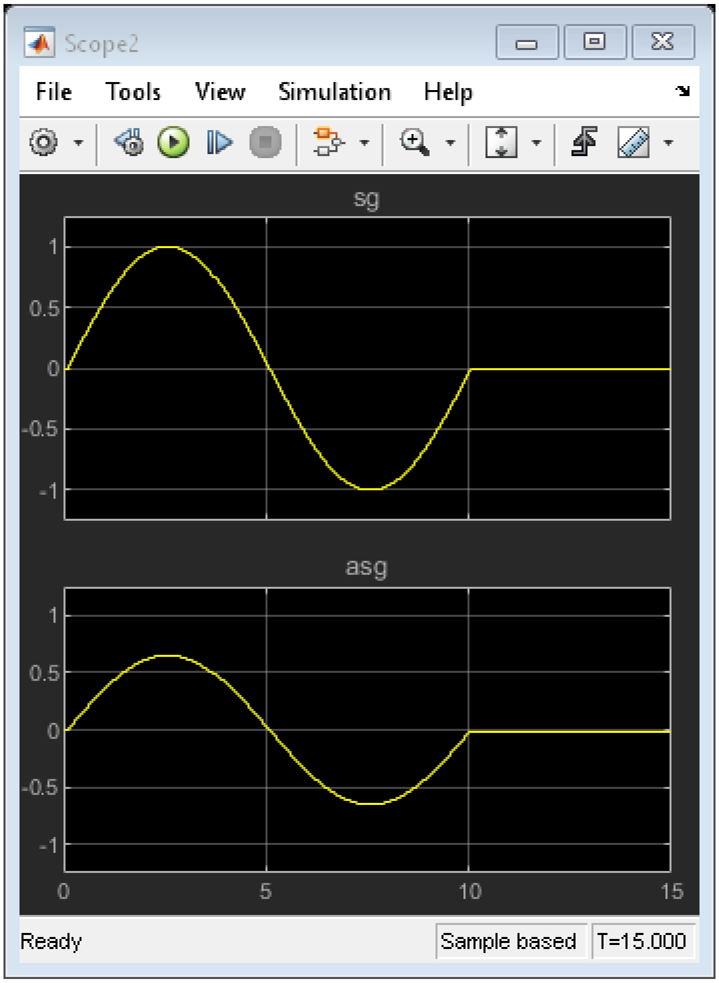


Step	Transition	Next Step
Initialize sg = 0;	1. true	ReducedSine ▼

ReducedSine
sg = sin(et*2*pi/10);
asg = Attenuate(sg);
	1. after(10,sec)	Stop ▼

Stop sg = 0;		

The test sequence produces signal `sg` and attenuated signal `asg`.



Signal Generation Functions

Some signal generation functions use the temporal operator `et`, which is the elapsed time of the test step in seconds. Scaling, rounding, and other approximations of argument values can affect function outputs. Common signal generation functions include:

Function	Syntax	Description	Example
`square`	`square(x)`	Represents a square wave output with a period of `1` and range –`1` to `1`. Within the interval `0 <= x < 1`, `square(x)` returns the value `1` for `0 <= x < 0.5`and –`1` for `0.5 <= x < 1`.	Output a square wave with a period of `10` sec: `square(et/10)`
`sawtooth`	`sawtooth(x)`	Represents a sawtooth wave output with a period of `1` and range –`1` to `1`. Within the interval `0 <= x < 1`, `sawtooth(x)` increases.	Output a sawtooth wave with a period of `10` sec: `sawtooth(et/10)`
`triangle`	`triangle(x)`	Represents a triangle wave output with a period of `1` and range –`1` to `1`. Within the interval `0 <= x < 0.5`, `triangle(x)` increases.	Output a triangle wave with a period of `10` sec: `triangle(et/10)`
`ramp`	`ramp(x)`	Represents a ramp signal of slope `1`, returning the value of the ramp at time `x`. `ramp(et)` effectively returns the elapsed time of the test step.	Ramp one unit for every 5 seconds of test step elapsed time: `ramp(et/5)`
`heaviside`	`heaviside(x)`	Represents a heaviside step signal, returning `0` for `x < 0` and `1` for `x >= 0`.	Output a heaviside signal after `5` seconds: `heaviside(et-5)`

Function	Syntax	Description	Example
`latch`	`latch(x)`	Saves the value of `x` at the first time `latch(x)` evaluates in a test step, and subsequently returns the saved value of `x`. Resets the saved value of `x` when the step exits. Reevaluates `latch(x)` when the step is next active.	Latch `b` to the value of `torque`: `b = latch(torque)`
`sin`	`sin(x)`	Returns the sine of `x`, where `x` is in radians.	A sine wave with a period of 10 sec: `sin(et*2*pi/10)`
`cos`	`cos(x)`	Returns the cosine of `x`, where `x` is in radians.	A cosine wave with a period of 10 sec: `cos(et*2*pi/10)`

`rand`	`rand`	Uniformly distributed pseudorandom values	Generate new random values for each simulation by declaring `rand` extrinsic with `coder.extrinsic`. Assign the random number to a local variable. For example:
			
			`coder.extrinsic('rand')`
			`nr = rand`
			`sg = a + (b-a)*nr`

Function	Syntax	Description	Example

`randn`	`randn`	Normally distributed pseudorandom values	Generate new random values for each simulation by declaring `randn` extrinsic with `coder.extrinsic`. Assign the random number to a local variable. For example:
			`coder.extrinsic('randn')`
			`nr = randn`
			`sg = nr*2`

`exp`	`exp(x)`	Returns the natural exponential function, e^x.	An exponential signal progressing at one tenth of the test step elapsed time: `exp(et/10)`

See Also

"Test Sequence and Assessment Syntax" on page 3-60 | Test Sequence

Related Examples

- "Assess Model Simulation Using verify Statements" on page 3-15
- "Actions and Transitions" on page 3-38

Programmatically Create a Test Sequence

This example shows how to create a test harness and test sequence using the programmatic interface. You create a test harness and a Test Sequence block, and author a test sequence to verify two functional attributes of a cruise control system.

Create a Test Harness Containing a Test Sequence Block

1. Load the model.

```
model = 'sltestCruiseChart';
load_system(fullfile(matlabroot,'examples','simulinktest','sltestCruiseChart'))
```

2. Create the test harness.

```
sltest.harness.create(model,'Name','Harness1',...
    'Source','Test Sequence')
sltest.harness.load(model,'Harness1');
set_param('Harness1','StopTime','15');
```

Author the Test Sequence

1. Add a local variable `endTest` and set the data type to `boolean`. You use `endTest` to transition between test steps.

```
sltest.testsequence.addSymbol('Harness1/Test Sequence','endTest',...
    'Data','Local');

sltest.testsequence.editSymbol('Harness1/Test Sequence','endTest',...
    'DataType','boolean');
```

2. Change the name of the step `Run` to `Initialize1`.

```
sltest.testsequence.editStep('Harness1/Test Sequence','Run',...
    'Name','Initialize1');
```

3. Add a step `BrakeTest`. `BrakeTest` checks that the cruise control disengages when the brake is applied. Add substeps defining the test scenario actions and verification.

```
sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
    'BrakeTest','Initialize1','Action','endTest = false;')

    % Add a transition from |Initialize1| to |BrakeTest|.
    sltest.testsequence.addTransition('Harness1/Test Sequence',...
        'Initialize1','true','BrakeTest')

    % This sub-step enables the cruise control and sets the speed.
    % |SetValuesActions| is the actions for BrakeTest.SetValues.
    setValuesActions = sprintf('CruiseOnOff = true;\nSpeed = 50;');
    sltest.testsequence.addStep('Harness1/Test Sequence',...
        'BrakeTest.SetValues','Action',setValuesActions)

    % This sub-step engages the cruise control.
    setCCActions = sprintf('CoastSetSw = true;');
    sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
        'BrakeTest.Engage','BrakeTest.SetValues','Action',setCCActions)

    % This step applies the brake.
    brakeActions = sprintf('CoastSetSw = false;\nBrake = true;');
    sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
        'BrakeTest.Brake','BrakeTest.Engage','Action',brakeActions)

    % This step verifies that the cruise control is off.
    brakeVerifyActions = sprintf('verify(engaged == false)\nendTest = true;');
    sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
        'BrakeTest.Verify','BrakeTest.Brake','Action',brakeVerifyActions)

    % Add transitions between steps.
    sltest.testsequence.addTransition('Harness1/Test Sequence',...
        'BrakeTest.SetValues','true','BrakeTest.Engage')
    sltest.testsequence.addTransition('Harness1/Test Sequence',...
        'BrakeTest.Engage','after(2,sec)','BrakeTest.Brake')
    sltest.testsequence.addTransition('Harness1/Test Sequence',...
        'BrakeTest.Brake','true','BrakeTest.Verify')
```

4. Add a step `Initialize2` to initialize component inputs. Add a transition from `BrakeTest` to `Initialize2`.

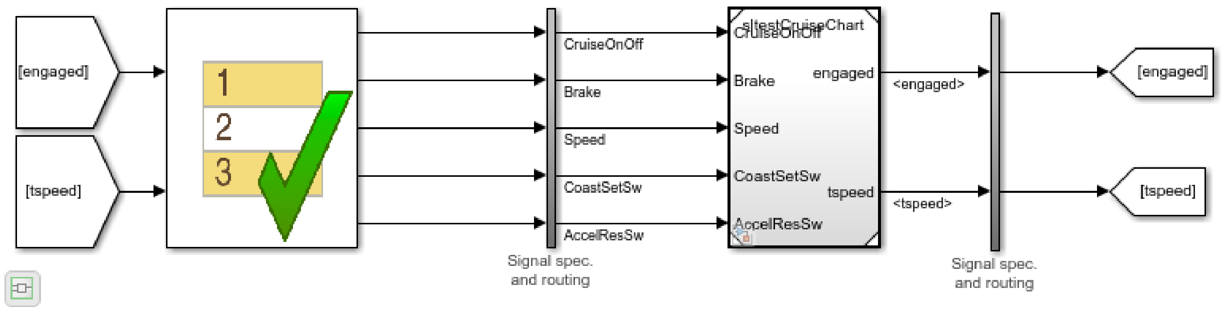
```
init2Actions = sprintf(['CruiseOnOff = false;\n'...
    'Brake = false;\n'...
    'Speed = 0;\n'...
    'CoastSetSw = false;\n'...
    'AccelResSw = false;']);
sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
    'Initialize2','BrakeTest','Action',init2Actions)
sltest.testsequence.addTransition('Harness1/Test Sequence',...
    'BrakeTest','endTest == true','Initialize2')
```

5. Add a step `LimitTest`. `LimitTest` checks that the cruise control disengages when the vehicle speed exceeds the high limit. Add a transition from the `Initialize2` step, and add sub-steps to define the actions and verification.

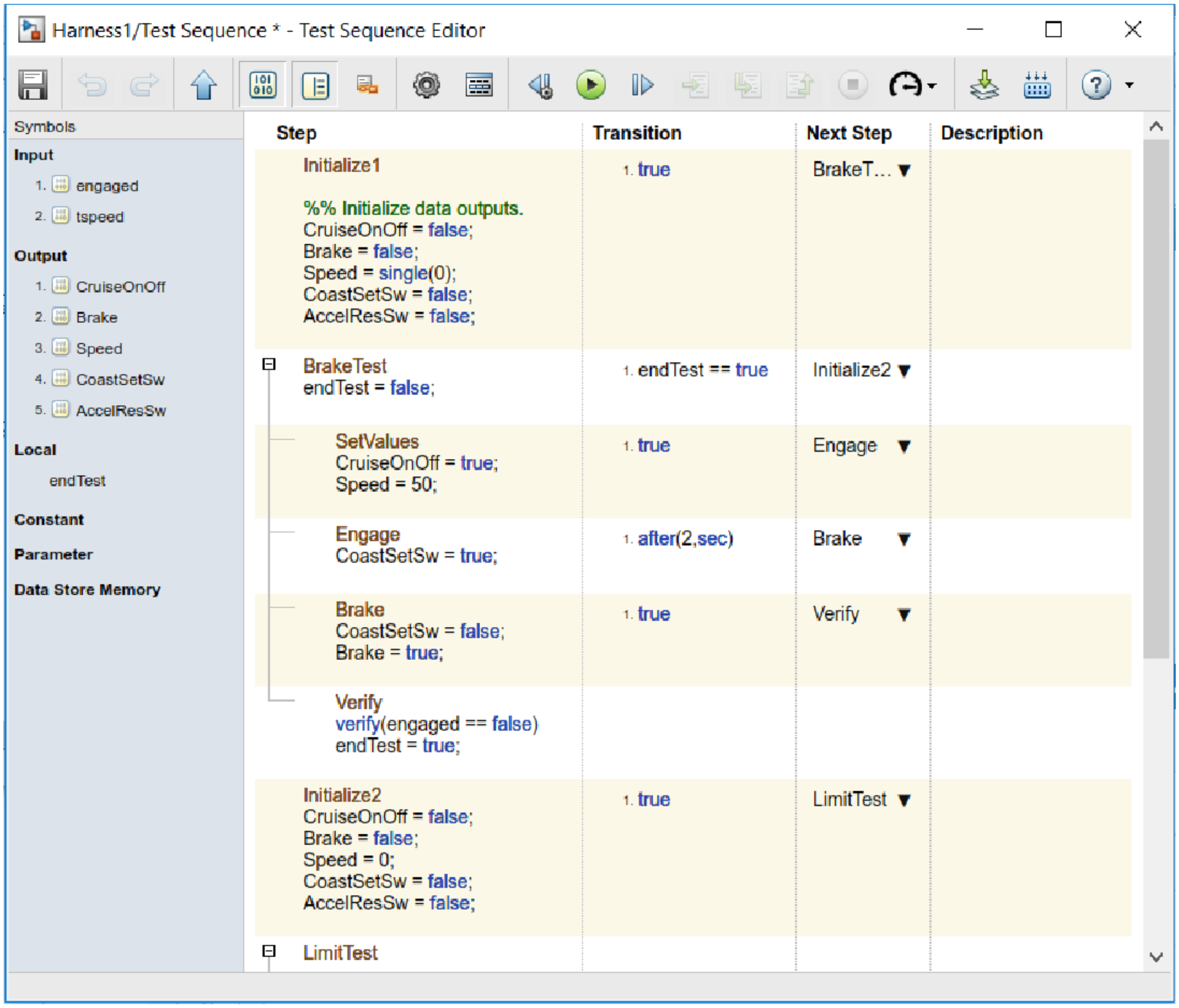
```
sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
    'LimitTest','Initialize2')
sltest.testsequence.addTransition('Harness1/Test Sequence',...
    'Initialize2','true','LimitTest')

    % Add a step to enable cruise control and set the speed.
    setValuesActions2 = sprintf('CruiseOnOff = true;\nSpeed = 60;');
    sltest.testsequence.addStep('Harness1/Test Sequence',...
        'LimitTest.SetValues','Action',setValuesActions2)

    % Add a step to engage the cruise control.
    setCCActions = sprintf('CoastSetSw = true;');
    sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
        'LimitTest.Engage','LimitTest.SetValues','Action',setCCActions)

    % Add a step to ramp the vehicle speed.
    sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
        'LimitTest.RampUp','LimitTest.Engage','Action','Speed = Speed + ramp(5*et);')

    % Add a step to verify that the cruise control is off.
    highLimVerifyActions = sprintf('verify(engaged == false)');
    sltest.testsequence.addStepAfter('Harness1/Test Sequence',...
        'LimitTest.VerifyHigh','LimitTest.RampUp','Action',highLimVerifyActions)

    % Add transitions between steps. The speed ramp transitions when the
    % vehicle speed exceeds 90.
    sltest.testsequence.addTransition('Harness1/Test Sequence',...
        'LimitTest.SetValues','true','LimitTest.Engage')
    sltest.testsequence.addTransition('Harness1/Test Sequence',...
        'LimitTest.Engage','true','LimitTest.RampUp')
    sltest.testsequence.addTransition('Harness1/Test Sequence',...
        'LimitTest.RampUp','Speed > 90','LimitTest.VerifyHigh')
```

Open the test harness to view the test sequence.

```
sltest.harness.open(model,'Harness1');
```



Double-click the Test Sequence block to open the editor and view the test sequence.



Step	Transition	Next Step	Description

Initialize1
 	
%% Initialize data outputs.
CruiseOnOff = false;
Brake = false;
Speed = single(0);
CoastSetSw = false;
AccelResSw = false;
	1. true	BrakeT...	

BrakeTest endTest = false;	1. endTest == true	Initialize2	

SetValues
CruiseOnOff = true;
Speed = 50;
	1. true	Engage	

Engage CoastSetSw = true;	1. after(2,sec)	Brake	

Brake
CoastSetSw = false;
Brake = true;
	1. true	Verify	

Verify
verify(engaged == false)
endTest = true;			

Initialize2
CruiseOnOff = false;
Brake = false;
Speed = 0;
CoastSetSw = false;
AccelResSw = false;
	1. true	LimitTest	

LimitTest			

Close the Test Harness and Model

```
sltest.harness.close(model,'Harness1');
close_system(model,0);
```

Test Sequence and Assessment Syntax

In this section...

This topic describes syntax used within Test Sequence and Test Assessment blocks. You use this syntax for test step actions, transitions, and assessments.

For information on using the command-line interface to create and edit test sequence steps, transitions, and data symbols, see the functions listed under **Test Sequences** on the "Test Scripts" page.

Test Sequence and Test Assessment blocks use MATLAB as the action language. You define actions, transitions, and assessments with assessment operators, temporal operators, transition operators, signal generation functions, logical operators, and relational operators. For example:

- To output a square wave with a period of `10` sec:

  ```
  square(et/10)
  ```
- To transition when `h` changes to `0`:

  ```
  hasChangedTo(h,0)
  ```
- To verify that x is greater than y:

  ```
  verify(x > y)
  ```

Assessment Statements

To verify simulation, stop simulation, and return verification results, use assessment statements.

Keyword	Statement Syntax	Description	Example

`verify`	`verify(expression)`
	
	`verify(expression, errorMessage)`
	
	`verify(expression, identifier, errorMessage)`
		Assesses a logical expression. Optional arguments label results in the Test Manager and diagnostic viewer.	`verify(x > y,...`
			`'SimulinkTest:greaterThan',...`
			`'x and y values are %d, %d',x,y)`

`assert`	`assert(expression)`
	
	`assert(expression, errorMessage)`
		Evaluates a logical expression. Failure stops simulation and returns an error. Optional arguments return an error message.	`assert(h == 0 && k == 0,...`
			`'h and k must initialize to 0')`

Syntax in the table uses these arguments:

`expression`

Logical statement assessed

Examples:

```
h > 0 && k == 0
```

`identifier`

Label applied to results in the Test Manager

Value: String of the form `aaa:bbb:...:zzz`, with at least two colon-separated MATLAB identifiers `aaa`, `bbb`, and `zzz`.

Examples:

```
'SimulinkTest:greaterThan'
```

`errorMessage`

Label applied to messages in the diagnostic viewer

Value: String

Examples:

```
'x and y values are %d, %d',x,y
```

Temporal Operators

To create an expression that evaluates the simulation time, use temporal operators. Variables used in signal conditions must be inputs, parameters, or constants in the Test Sequence block.

Operator	Syntax	Description	Example
`et`	`et(TimeUnits)`	The elapsed time of the test step in `TimeUnits`. Omitting `TimeUnits` returns the value in seconds.	The elapsed time of the test sequence step in milliseconds: `et(msec)`
`t`	`t(TimeUnits)`	The elapsed time of the simulation in `TimeUnits`. Omitting `TimeUnits` returns the value in seconds.	The elapsed time of the simulation in microseconds: `t(usec)`
`after`	`after(n, TimeUnits)`	Returns `true` if `n` specified units of time in `TimeUnits` elapse since the beginning of the current test step.	After 4 seconds: `after(4,sec)`
`before`	`before(n, TimeUnits)`	Returns `true` until `n` specified units of time in `TimeUnits` elapse, beginning with the current test step.	Before 4 seconds: `before(4,sec)`

Operato r	Syntax	Description	Example
`duration`	`ElapsedTime = duration (Condition, TimeUnits)`	Returns `ElapsedTime` in `TimeUnits` for which `Condition` has been `true`. `ElapsedTime` is reset when the test step is re-entered or when `Condition` is no longer `true`.	Return `true` if the time in milliseconds since `Phi > 1` is greater than 550: `duration(Phi>1,msec) > 550`

Syntax in the table uses these arguments:

`TimeUnits`

The units of time

Value: `sec|msec|usec`

Examples:

```
msec
```

`Condition`

Logical expression triggering the operator. Variables used in `duration` can be inputs, parameters, or constants, with at most one local or output data.

Examples:

```
u > 0
x <= 1.56
```

Transition Operators

To create expressions that evaluate signal events, use transition operators. Common transition operators include:

Operator	Syntax	Description	Example
`hasChanged`	`hasChanged(u)`	Returns `true` if `u` changes in value since the beginning of the test step, otherwise returns `false`. `u` must be an input data symbol.	Transition when `h` changes: `hasChanged(h)`
`hasChangedFrom`	`hasChangedFrom(u,A)`	Returns true if `u` changes from the value `A`, otherwise returns false. `u` must be an input data symbol.	Transition when `h` changes from `1`: `hasChangedFrom(h,1)`
`hasChangedTo`	`hasChangedTo(u,B)`	Returns true if `u` changes to the value `B`, otherwise returns false. `u` must be an input data symbol.	Transition when `h` changes to `0`: `hasChangedTo(h,0)`

Signal Generation Functions

Some signal generation functions use the temporal operator `et`, which is the elapsed time of the test step in seconds. Scaling, rounding, and other approximations of argument values can affect function outputs. Common signal generation functions include:

Function	Syntax	Description	Example
`square`	`square(x)`	Represents a square wave output with a period of `1` and range –`1` to `1`. Within the interval `0 <= x < 1`, `square(x)` returns the value `1` for `0 <= x < 0.5`and –`1` for `0.5 <= x < 1`.	Output a square wave with a period of `10` sec: `square(et/10)`
`sawtooth`	`sawtooth(x)`	Represents a sawtooth wave output with a period of `1` and range –`1` to `1`. Within the interval `0 <= x < 1`, `sawtooth(x)` increases.	Output a sawtooth wave with a period of `10` sec: `sawtooth(et/10)`
`triangle`	`triangle(x)`	Represents a triangle wave output with a period of `1` and range –`1` to `1`. Within the interval `0 <= x < 0.5`, `triangle(x)` increases.	Output a triangle wave with a period of `10` sec: `triangle(et/10)`
`ramp`	`ramp(x)`	Represents a ramp signal of slope `1`, returning the value of the ramp at time `x`. `ramp(et)` effectively returns the elapsed time of the test step.	Ramp one unit for every 5 seconds of test step elapsed time: `ramp(et/5)`
`heaviside`	`heaviside(x)`	Represents a heaviside step signal, returning `0` for `x < 0` and `1` for `x >= 0`.	Output a heaviside signal after `5` seconds: `heaviside(et-5)`

Function	Syntax	Description	Example
`latch`	`latch(x)`	Saves the value of `x` at the first time `latch(x)` evaluates in a test step, and subsequently returns the saved value of `x`. Resets the saved value of `x` when the step exits. Reevaluates `latch(x)` when the step is next active.	Latch `b` to the value of `torque`: `b = latch(torque)`
`sin`	`sin(x)`	Returns the sine of `x`, where `x` is in radians.	A sine wave with a period of 10 sec: `sin(et*2*pi/10)`
`cos`	`cos(x)`	Returns the cosine of `x`, where `x` is in radians.	A cosine wave with a period of 10 sec: `cos(et*2*pi/10)`

`rand`	`rand`	Uniformly distributed pseudorandom values	Generate new random values for each simulation by declaring `rand` extrinsic with `coder.extrinsic`. Assign the random number to a local variable. For example:
			
			`coder.extrinsic('rand')`
			`nr = rand`
			`sg = a + (b-a)*nr`

Function	Syntax	Description	Example

`randn`	`randn`	Normally distributed pseudorandom values	Generate new random values for each simulation by declaring `randn` extrinsic with `coder.extrinsic`. Assign the random number to a local variable. For example:
			`coder.extrinsic('randn')`
			`nr = randn`
			`sg = nr*2`

`exp`	`exp(x)`	Returns the natural exponential function, e^x.	An exponential signal progressing at one tenth of the test step elapsed time: `exp(et/10)`

Logical Operators

You can use logical connectives in actions, transitions, and assessments. In these examples, `p` and `q` represent Boolean signals or logical expressions.

Operation	Syntax	Description	Example
Negation	`~p`	not `p`	`verify(~p)`
Conjunction	`p && q`	`p` and `q`	`verify(p && q)`
Disjunction	`p || q`	`p` or `q`	`verify(p || q)`
Implication	`~p || q`	if `p`, `q`. Logically equivalent to implication `p` → `q`.	`verify(~p || q)`
Biconditional	`(p && q) || (~p && ~q)`	`p` and `q`, or not `p` and not `q`. Logically equivalent to biconditional `p` ↔ `q`.	`verify((p && q) || (~p && ~q))`

Relational Operators

You can use relational operators in actions, transitions, and assessments. In these examples, x and y represent numeric-type variables.

Using == or ~= operators in a verify statement returns a warning when comparing floating-point data. Consider the precision limitations associated with floating-point numbers when implementing verify statements. See “Floating-Point Numbers” (MATLAB). If you use floating-point data, consider defining a tolerance for the assessment. For example, instead of verify(x == 5), verify x within a tolerance of 0.001:

```
verify(abs(x-5) < 0.001)
```

Operator and Syntax	Description	Example
x > y	Greater than	verify(x > y)
x < y	Less than	verify(x < y)
x >= y	Greater than or equal to	verify(x >= y)
x <= y	Less than or equal to	verify(x <= y)
x == y	Equal to	verify(x == y)
x ~= y	Not equal to	verify(x ~= y)

See Also

Related Examples

- “Assess Model Simulation Using verify Statements” on page 3-15
- “Actions and Transitions” on page 3-38
- “Signal Generation Functions” on page 3-47
- “Programmatically Create a Test Sequence” on page 3-55

Debug a Test Sequence

In this section...

You can debug a test sequence using tools in the Test Sequence Editor. Debugging involves setting breakpoints to stop simulation, observing data and test sequence progression, and manually stepping through test steps. You can try these features using the model `sltestTestSeqDebuggingExample`. To open the model, enter

```
cd(fullfile(docroot,'toolbox','sltest','examples'))
open_system('sltestTestSeqDebuggingExample')
```

Save a copy of the model to a writable location on the MATLAB path. Double-click the Test Sequence block to open the Test Sequence Editor.

View Test Step Execution During Simulation

By default, simulation animates the test sequence by highlighting active steps and transitions. Observing test step execution can help you debug, particularly when manually stepping through the test sequence. Adjust the animation speed using the **Change Animation Speed** button in the toolbar.

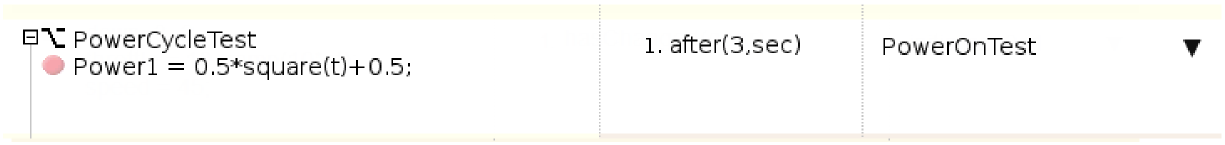
Animation speed affects simulation speed. If you slow down animation speed for debugging, return the speed to **Fast** or **Lightning Fast** when you finish debugging to avoid slowing your simulation. If you do not need the test step highlights and want the fastest simulation, choose **None**.

Set Breakpoints to Enable Debugging

You enable debugging for a test sequence by adding one or more breakpoints. Breakpoints halt simulation every time the test step is evaluated. Therefore, breakpoints on some test steps, such as **When decomposition** parent steps, halt simulation repeatedly because the step is evaluated repeatedly. When simulation halts, you can view data used in the test sequence to investigate the sequence simulation behavior.

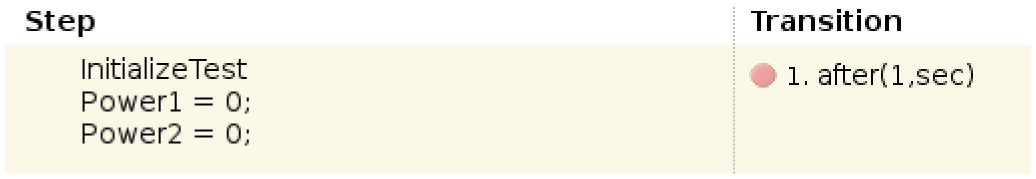
You can add breakpoints to test step actions or transitions:

- To add a breakpoint to a test step action, right-click the test step and select **Break while executing step**.



```
PowerCycleTest
Power1 = 0.5*square(t)+0.5;
```

- To add a breakpoint to a test step transition, right-click the test step transition and select **Break when transition taken**.



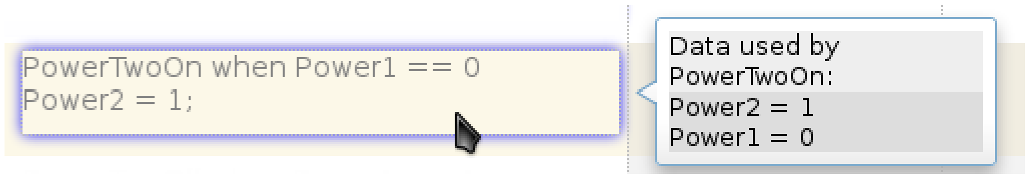
Step	Transition

InitializeTest
Power1 = 0;
Power2 = 0;
	● 1. after(1,sec)

The editor displays a breakpoint marker. After adding breakpoints, simulate the test sequence by clicking **Run**.

View Data Values During Simulation

If the simulation pauses (for example, at a breakpoint), you can view the status of data used in a test step by hovering over the test step. The data values at the current simulation time display next to the test sequence cell.



```
PowerTwoOn when Power1 == 0
Power2 = 1;
```

Data used by PowerTwoOn:
Power2 = 1
Power1 = 0

Note If you advance the simulation to another stop (for example, using the keyboard shortcuts), the data display does not update. Move off the test step and then hover over the step again to refresh the values.

Step Through Simulation

When simulation halts, you can step through the test sequence using the toolbar buttons. Also see “Debugging and Breakpoints Keyboard Shortcuts” (Simulink).

Objective	Details	Toolbar Button
Simulate until breakpoint	Simulation runs until the next breakpoint	▶
Step forward through simulation time	Simulation advances one simulation step	⏵
Step forward through test step actions and transitions	Simulation advances by each step of a test sequence, with pauses at actions and transitions. Does not step into a function call.	→
Step in to a test step group or called function	Simulation advances into the substeps of a parent step and executes each action and transition. Steps into a function call.	↳
Step out of a test step group or called function	Simulation advances through the remaining substeps of a parent step and then out to the parent step hierarchy level. Also finishes execution of a function call.	⮥

See Also

“Test Sequence Editor” on page 3-24 | Test Sequence

Test Downshift Points of a Transmission Controller

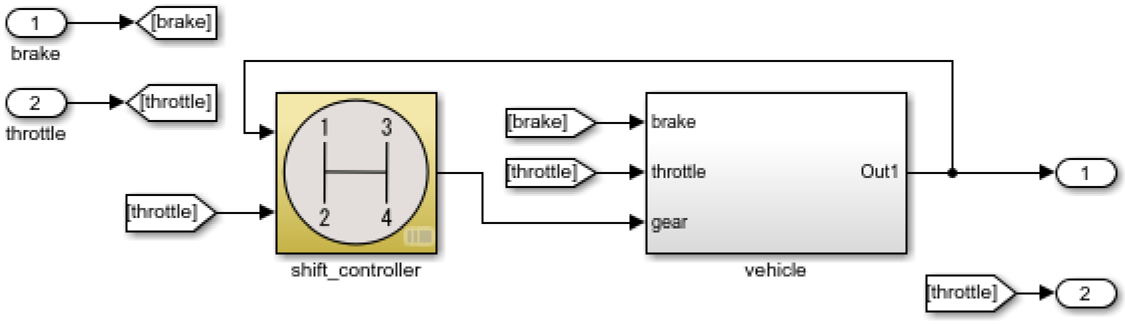
This example demonstrates how to test a transmission shift logic controller using test sequences and test assessments.

The Model and Controller

This example uses a simplified drivetrain system arranged in a controller-plant configuration. The objective of the example is to test the transmission controller in isolation, ensuring that it downshifts correctly.

The Test

The controller should downshift between each of its gear ratios in response to a ramped throttle application. The test inputs hold vehicle speed constant while ramping the throttle. The Test Assessment block includes requirements-based assessments of the controller performance.

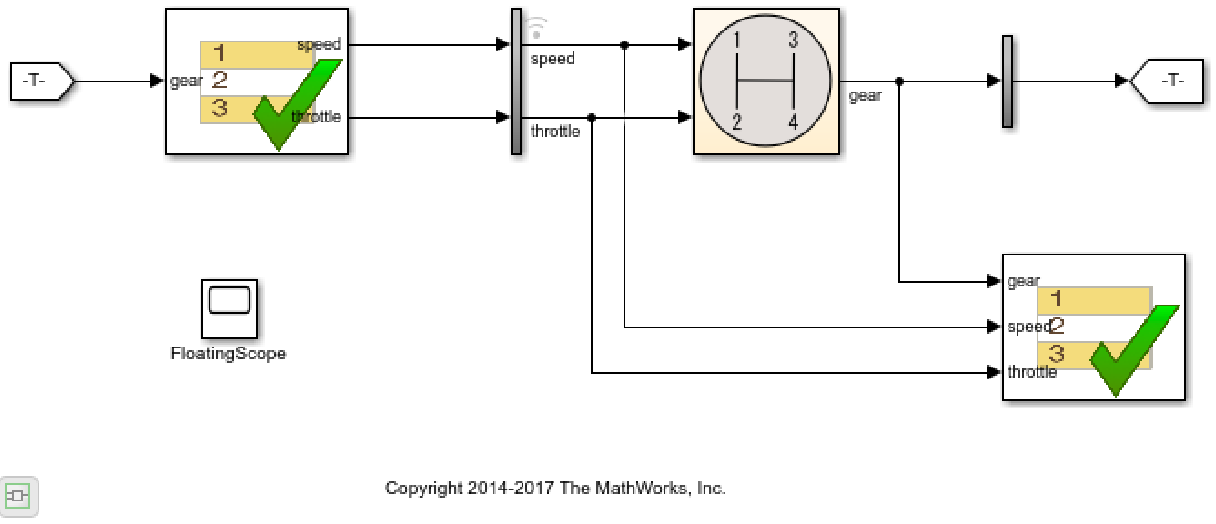


Testing Downshift Points of a Transmission Controller

Copyright 2014-2017 The MathWorks, Inc.

Open the Test Harness

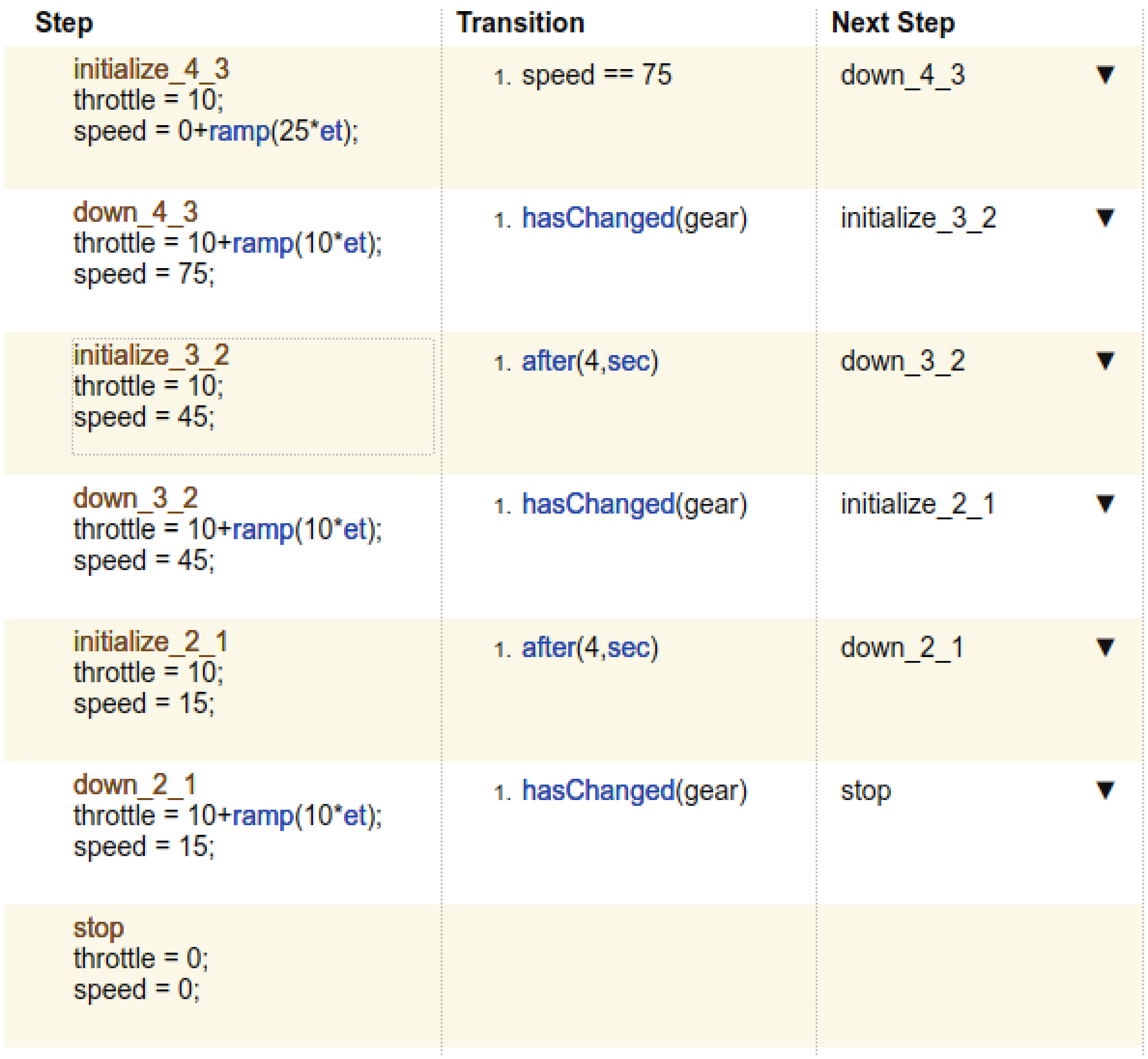
Click the badge on the subsystem `shift_controller` and open the test harness `controller_harness`. `shift_controller` is connected to a Test Sequence block and a Test Assessment block.



The Test Sequence

Double-click the Test Sequence block to open the test sequence editor.

The test sequence begins by ramping speed to 75 to initialize the controller to fourth gear. Throttle is then ramped at constant speed until a gear change. Subsequent initialization and downshifts execute. After the change to first gear, the test sequence stops.



Step	Transition	Next Step

initialize_4_3
throttle = 10;
speed = 0+ramp(25*et);
	1. speed == 75	down_4_3

down_4_3
throttle = 10+ramp(10*et);
speed = 75;
	1. hasChanged(gear)	initialize_3_2

initialize_3_2
throttle = 10;
speed = 45;
	1. after(4,sec)	down_3_2

down_3_2
throttle = 10+ramp(10*et);
speed = 45;
	1. hasChanged(gear)	initialize_2_1

initialize_2_1
throttle = 10;
speed = 15;
	1. after(4,sec)	down_2_1

down_2_1
throttle = 10+ramp(10*et);
speed = 15;
	1. hasChanged(gear)	stop

stop
throttle = 0;
speed = 0;		

Test Assessments for the Controller

This example tests the following conditions:

- Speed shall never be negative.

- Gear shall always be positive.
- Throttle shall be between 0% and 100%.
- The shift controller shall keep the vehicle speed below specified maximums in each of the first three gears.

Open the Test Assessment block. The `assert` statements correspond to the first three conditions. If the controller violates one of the assertions, the simulation fails.

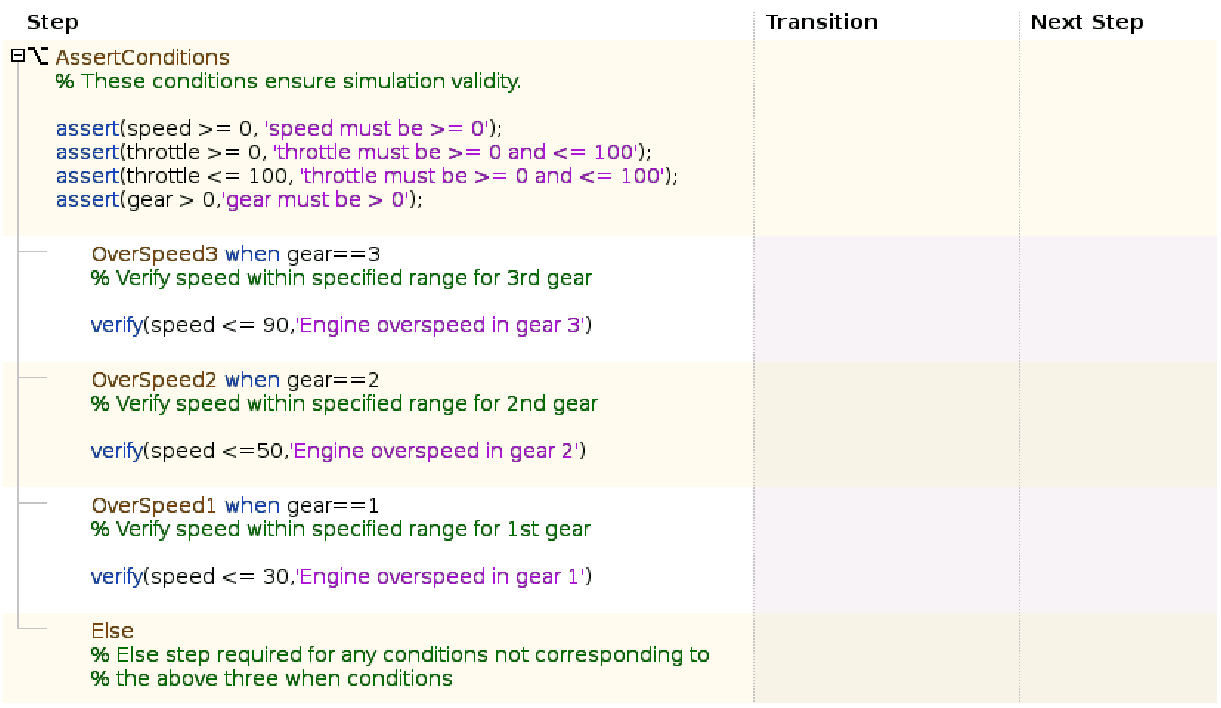
```
assert(speed >= 0, 'speed must be >= 0');
assert(throttle >= 0, 'throttle must be >= 0 and <= 100');
assert(throttle <= 100, 'throttle must be >= 0 and <= 100');
assert(gear > 0,'gear must be > 0');
```

The last condition is checked by three individual `verify` statements corresponding to the maximum speed in gears 1, 2, and 3:

- The controller shall not let the vehicle speed exceed 90 in gear 3.
- The controller shall not let the vehicle speed exceed 50 in gear 2.
- The controller shall not let the vehicle speed exceed 30 in gear 1.

The `verify` statements are contained in a When decomposition sequence. The active When decomposition step is determined by on signal conditions defined in the **Step** column, with each condition preceded by the `when` operator. The last step `Else` covers any undefined condition and does not use a `when` declaration. For more information on When decomposition, see Test Sequence Editor.

```
OverSpeed3 when gear==3
verify(speed <= 90,'Engine overspeed in gear 3')

OverSpeed2 when gear==2
verify(speed <= 50,'Engine overspeed in gear 2')

OverSpeed1 when gear==1
verify(speed <= 30,'Engine overspeed in gear 1')
```



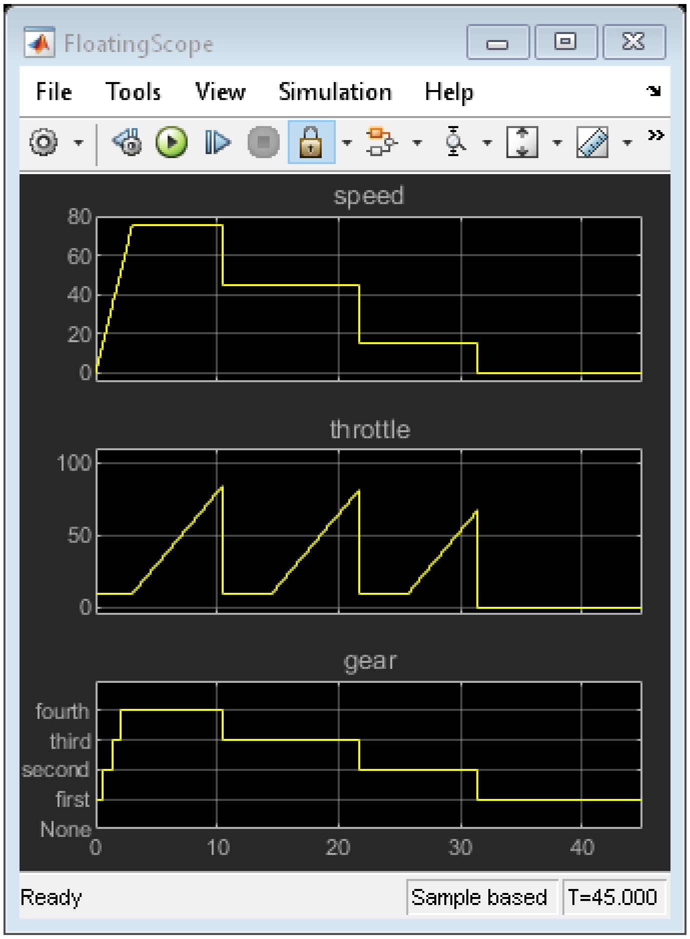
Step	Transition	Next Step

AssertConditions
% These conditions ensure simulation validity.
 	
assert(speed >= 0, 'speed must be >= 0');
assert(throttle >= 0, 'throttle must be >= 0 and <= 100');
assert(throttle <= 100, 'throttle must be >= 0 and <= 100');
assert(gear > 0,'gear must be > 0');		

OverSpeed3 when gear==3 % Verify speed within specified range for 3rd gear verify(speed <= 90,'Engine overspeed in gear 3')		
OverSpeed2 when gear==2 % Verify speed within specified range for 2nd gear verify(speed <=50,'Engine overspeed in gear 2')		
OverSpeed1 when gear==1 % Verify speed within specified range for 1st gear verify(speed <= 30,'Engine overspeed in gear 1')		
Else % Else step required for any conditions not corresponding to % the above three when conditions		

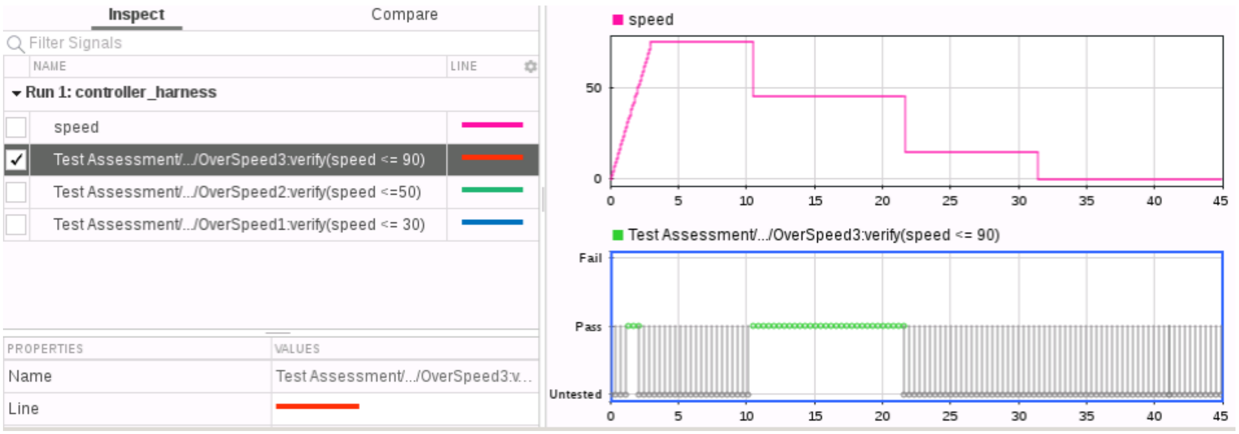
Testing the Controller

Simulating the test harness demonstrates the progressive throttle ramp at each test step and the corresponding downshifts. The controller passes all of the assessments in the Test Assessment block.



View the Results

Click the Simulation Data Inspector button in the test harness toolstrip to view the results. You can compare the `speed` signal to the `verify` statement outputs.


Inspect
Compare
Filter Signals
NAME
LINE
Run 1: controller_harness
speed
Test Assessment/.../OverSpeed3:verify(speed <= 90)
Test Assessment/.../OverSpeed2:verify(speed <=50)
Test Assessment/.../OverSpeed1:verify(speed <= 30)
PROPERTIES
VALUES
Name
Test Assessment/.../OverSpeed3:v...
Line
speed
Test Assessment/.../OverSpeed3:verify(speed <= 90)
Fail
Pass
Untested

View Graphical Results From Model Verification Library

Simulink® Test™ outputs graphical results of the Model Verification block library so you can use the Test Manager or Simulation Data Inspector to see when your test assessments pass and fail.

In addition to warnings or stop-simulation behavior, the graphical results show the block evaluation results during simulation. Viewing Model Verification block results graphically helps you to:

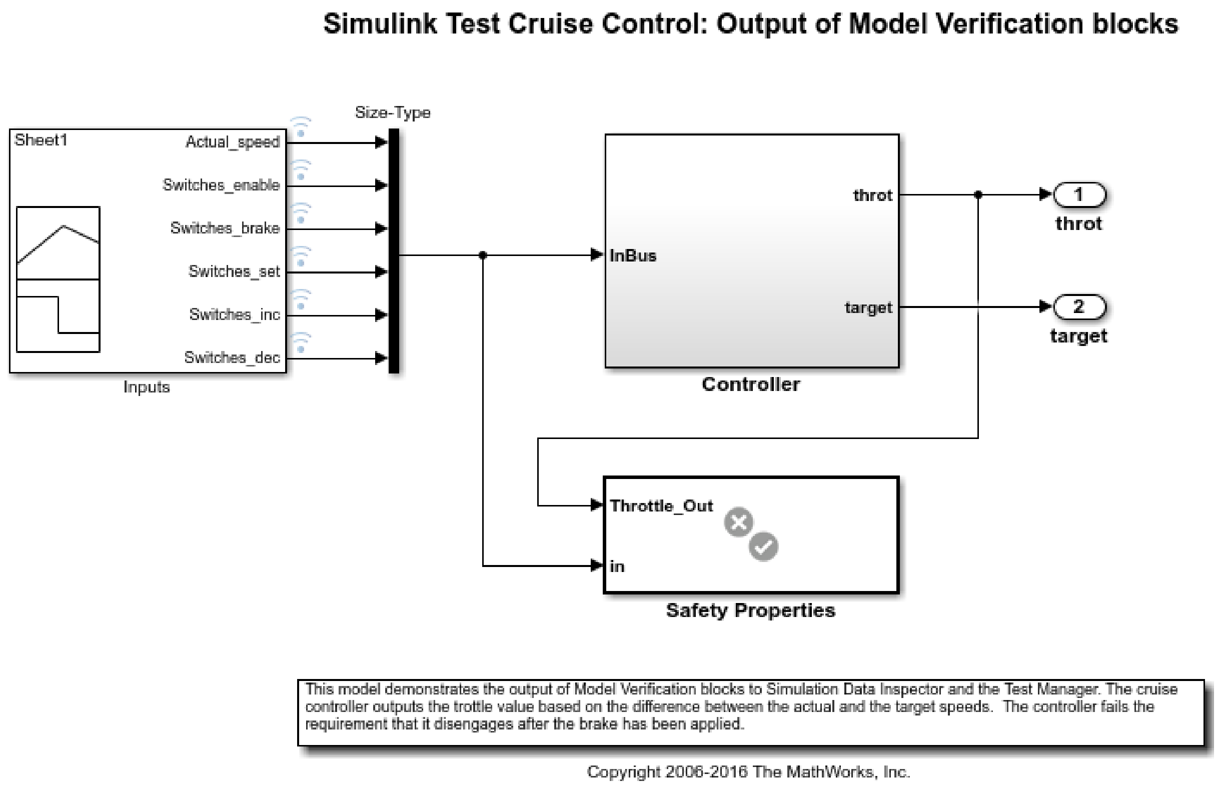
- Determine the time step when a failure occurs.
- Debug the model by comparing the verification result with relevant signals.
- Trace failures from the results to the model.

This example shows how to view outputs from Model Verification blocks in the Test Manager or Simulation Data Inspector.

Open the Model

The model contains a verification subsystem `Safety Properties` that uses an Assertion block to check whether the system disengages if the brake has been applied for three time steps. The verification subsystem also uses Simulink® Design Verifier™ blocks.

```
open_system(fullfile(matlabroot,'examples','simulinktest',...
    'sltestCruiseControlDefective'))
```



This model demonstrates the output of Model Verification blocks to Simulation Data Inspector and the Test Manager. The cruise controller outputs the trottle value based on the difference between the actual and the target speeds. The controller fails the requirement that it disengages after the brake has been applied.

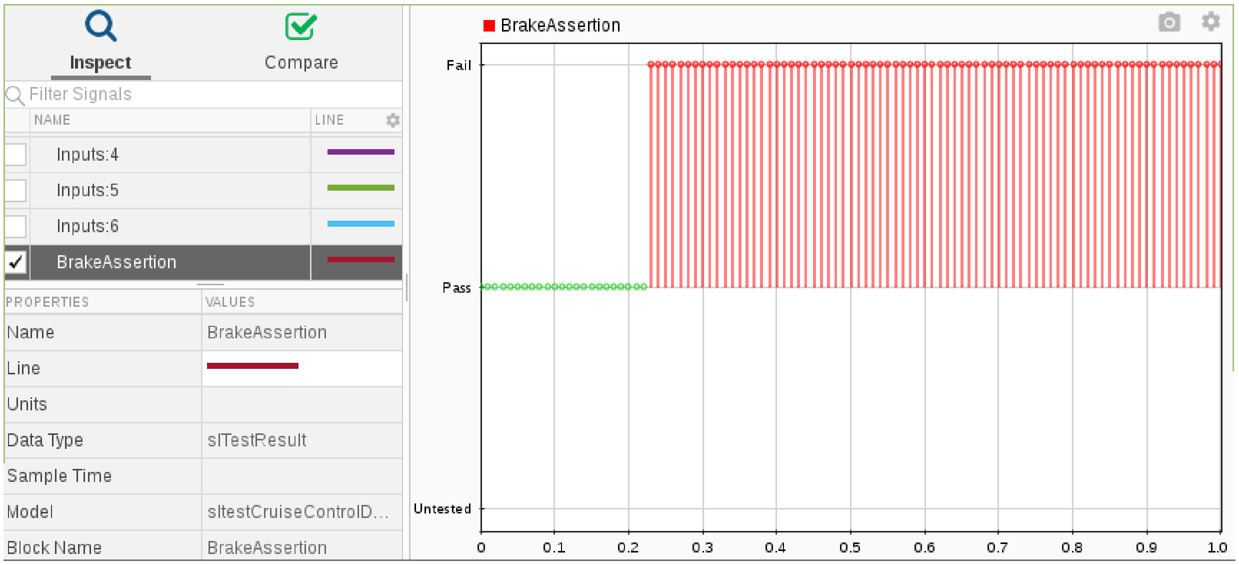
Copyright 2006-2016 The MathWorks, Inc.

Simulate the Model and View Results in SDI

```
sim('sltestCruiseControlDefective')
```

After the simulation completes, open SDI. The results show that the assertion failed at 0.23 seconds.

```
Simulink.sdi.view
```

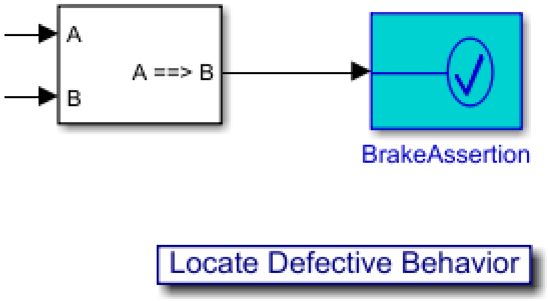


NAME	LINE
Inputs:4	
Inputs:5	
Inputs:6	
✓ BrakeAssertion	

PROPERTIES	VALUES
Name	BrakeAssertion
Line	
Units	
Data Type	slTestResult
Sample Time	
Model	sltestCruiseControlD...
Block Name	BrakeAssertion

Highlight Assertion Block in the Model

To find the assertion block in the model, right-click **BrakeAssertion** in SDI and select **Highlight in Model**. The block is highlighted in the verification subsystem.



See Also

`sltest.getAssessments`

Assess Temporal Logic Using Temporal Assessments

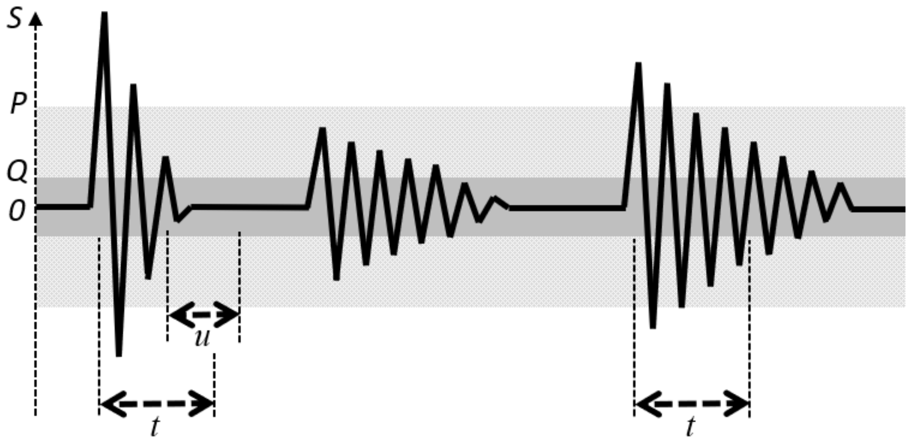
Hybrid systems with discrete and continuous time behavior can require complex timing-dependent signal logic. Simulink Test enables you to assess model timing and event ordering by authoring and including temporal assessments with test cases in the Test Manager.

To work with temporal assessments:

1. Select an assessment template
2. Enter assessment conditions:
   - Map symbols to model elements such as signals or to a time series or constant
   - View assessment summary
3. Run the test case
4. Use the results to assess the system under test (SUT) against your requirements

As an example, consider a forced oscillation damping problem that has this requirement:

For a signal *S*, if the signal magnitude exceeds a value *P*, then within *t* seconds, it must settle below a value *Q* and stay below *Q* for *u* seconds.

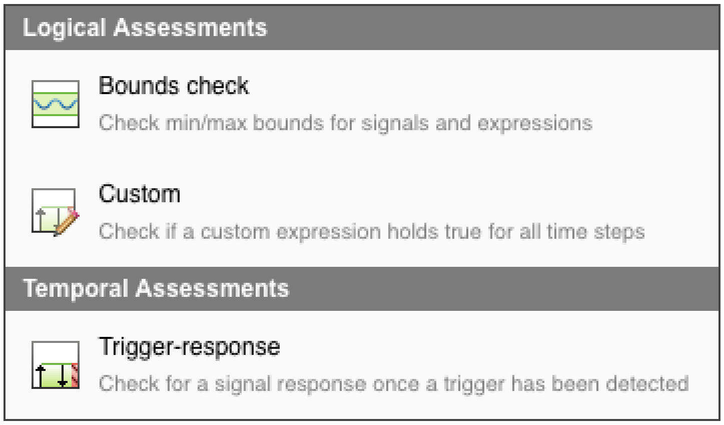


Create a Temporal Assessment

To create a temporal assessment:

1. Create or open a test case.
2. Navigate to the **Logical and Temporal Assessments** Editor.
3. Click **Add Assessment**. These assessment templates are available:

   - **Bounds Check –** Check maximum and minimum bounds for signals and expressions
   - **Trigger-Response –** Check for a signal response when a trigger is detected
   - **Custom –** Check if a logical expression holds true for all time steps

   Logical Assessments

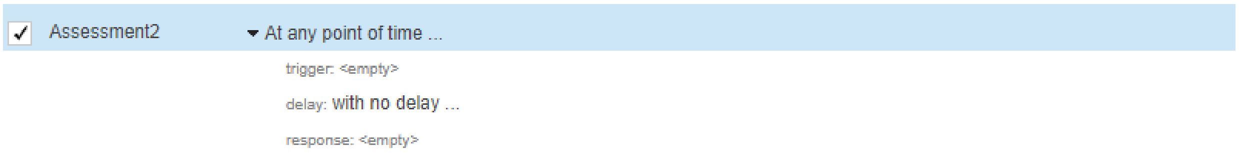
   Bounds check
   Check min/max bounds for signals and expressions

   Custom
   Check if a custom expression holds true for all time steps

   Temporal Assessments

   Trigger-response
   Check for a signal response once a trigger has been detected

   For this example, select **Trigger-Response**.



✓ Assessment2 At any point of time ...

trigger: <empty>

delay: with no delay ...

response: <empty>

The Trigger-Response template appears. To finish creating the assessment, you define temporal assessment conditions in the context of the SUT.

Define Temporal Assessment Conditions

A Trigger-response assessment requires:

- `Trigger` parameter

- `Response` parameter
- Optional `Delay` parameter

You can enter conditional statements as the trigger and response conditions. For the forced oscillation damping problem:

1. Select *whenever is true* as the trigger and enter `abs(S) > P` as the `condition`. The trigger condition is the condition pattern after which the response signal is evaluated. The response condition is triggered when the magnitude of signal `S` exceeds value `P`.
2. Select *must stay true for at least* as the response and enter `abs(S) < Q` and `u` as the `condition` and `min-time` respectively. The response condition describes the behavior of the SUT in response to the trigger condition. The response condition is that the magnitude of signal `S` must settle below value `Q` and stay below `Q` for at least `u` seconds.
3. Select *with a delay of at most* as the delay type and set `t` as the `max-time` parameter. The delay is an optional time interval starting from a time reference parameter to the point where the response condition is expected to be satisfied. The delay is at most `t` seconds.

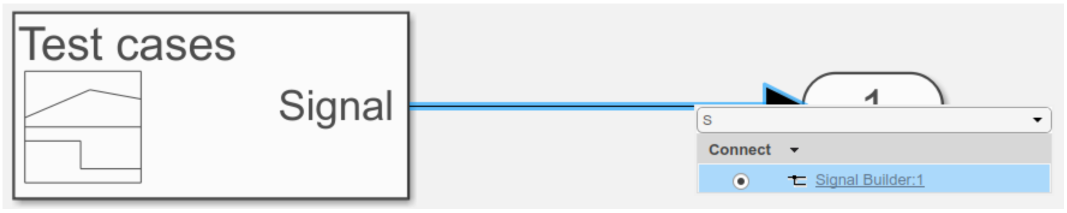
All time units are seconds.

When you add a symbol as part of a temporal assessment parameter in the Assessments editor, it is added to the list of symbols as an unresolved symbol. Resolve symbols by using the **Symbols** pane in the Assessments Editor.

Resolve Assessment Parameter Symbols

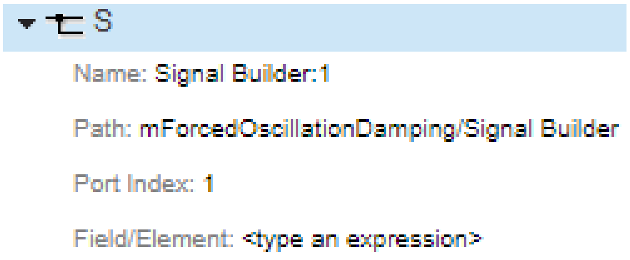
To resolve a symbol, right-click the symbol. Two options are available:

1. **Map to model element –** Use the mapping dialog box to map symbols to a signal/parameter/block in the SUT.



Select a symbol to map from the drop-down list at the top of the mapping dialog.

After you finish mapping symbols to model elements, the **Symbols** pane displays metadata that corresponds to the model element.



S

Name: Signal Builder:1

Path: mForcedOscillationDamping/Signal Builder

Port Index: 1

Field/Element: <type an expression>

Signals mapped to a symbol used by an assessment in the editor are logged when you run the test case.

If you map a bus or an array to a symbol, use the **Field/Element** row in the **Symbols** pane to select a scalar signal from the bus or array. For example:

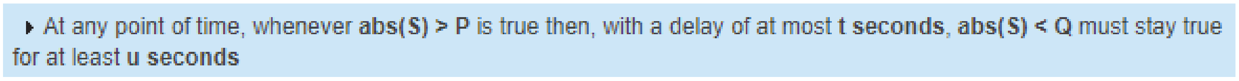
- If you want to map a symbol to a bus signal containing a bus element `fieldA`, enter `.fieldA`.
- If you want to map a symbol to the signal element corresponding to index (5,5) in a signal array, enter `(5,5)`.
- You can combine both expressions as `.fieldA(5,5)`.

2 **Map to expression –** Assign a scalar constant value or time series object to a symbol.

You can use the simulation output as a variable to map symbols to signals. For example, entering `sltest_simout.logsout.get('mySignal')` is equivalent to using **Map to model element** to map symbols to a signal `mySignal`. See “Test-Case Level Callbacks” on page 7-99 for more information.

Review Temporal Assessment Summary

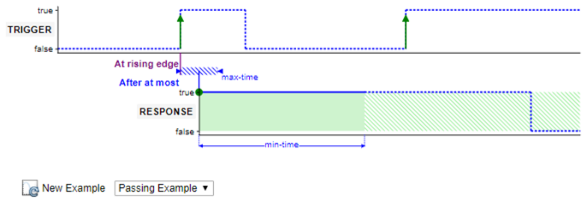
After you enter the assessment parameters, click the arrow to the left of the Assessment description to view the assessment summary.

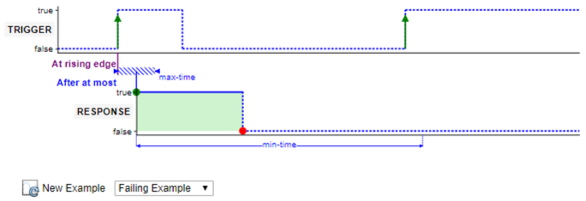


At any point of time, whenever **abs(S) > P** is true then, with a delay of at most **t seconds**, **abs(S) < Q** must stay true for at least **u seconds**

The **Visual Representation** pane provides a graphical illustration of a passing case for the assessment.

View Passing and Failing cases for the assessment by clicking the Explore Pattern icon. Select the type of case you want to view (passing or failing) from the drop-down list and click to view different passing and failing cases.



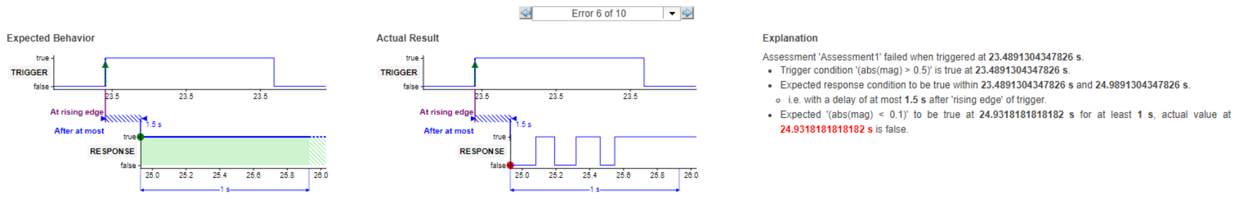


Evaluate the SUT

Run the test case to evaluate the SUT. Temporal assessments are evaluated after simulation using logged signal data. Use the test case results to review the SUT against your requirements.

View Assessment Results

View the results of the assessment evaluation from the **Results and Artifacts** pane of the Test Manager. Select the test case and select the assessment in the **Results** tree to open a new **Assessment Result** tab. Simulink Test evaluates the assessment and displays the expected behavior and the actual result of the assessment execution with a description of the assessment failures at different time steps.



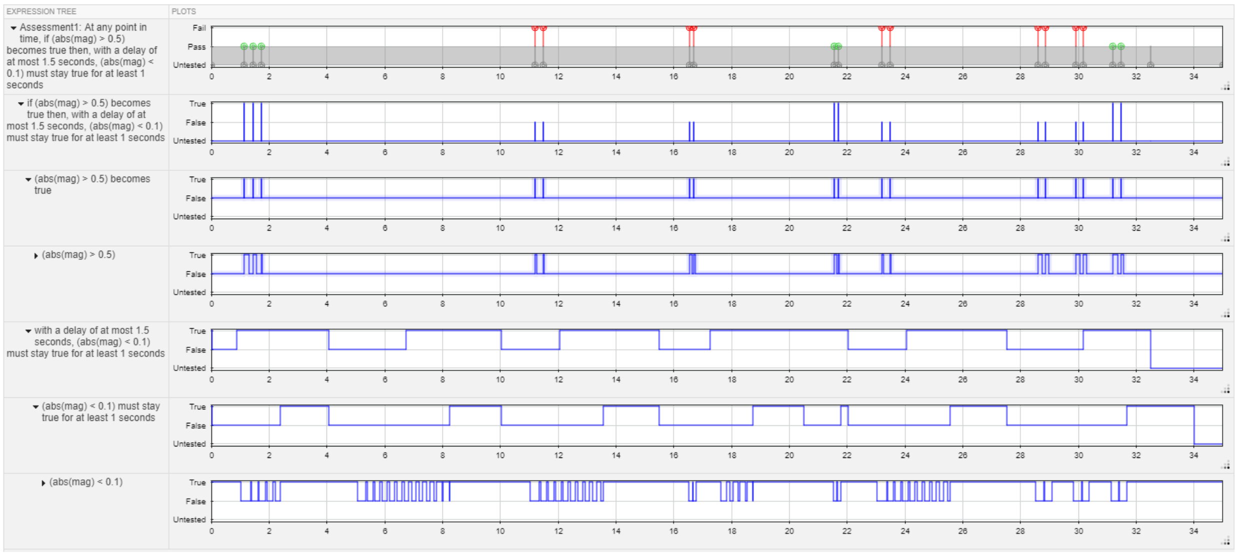
Explanation

Assessment 'Assessment1' failed when triggered at **23.4891304347826 s**.

- Trigger condition '(abs(mag) > 0.5)' is true at **23.4891304347826 s**.
- Expected response condition to be true within **23.4891304347826 s** and **24.9891304347826 s**.
  - i.e. with a delay of at most **1.5 s** after 'rising edge' of trigger.
- Expected '(abs(mag) < 0.1)' to be true at **24.9318181818182 s** for at least **1 s**, actual value at **24.9318181818182 s** is false.

Investigate the SUT behavior using the and buttons and the textual descriptions at points of failure.

For a more detailed investigation, expand the Expression Tree to view results for every individual element of the assessment.


EXPRESSION TREE
PLOTS
Assessment1: At any point in time, if (abs(mag) > 0.5) becomes true then, with a delay of at most 1.5 seconds, (abs(mag) < 0.1) must stay true for at least 1 seconds
if (abs(mag) > 0.5) becomes true then, with a delay of at most 1.5 seconds, (abs(mag) < 0.1) must stay true for at least 1 seconds
(abs(mag) > 0.5) becomes true
(abs(mag) > 0.5)
with a delay of at most 1.5 seconds, (abs(mag) < 0.1) must stay true for at least 1 seconds
(abs(mag) < 0.1) must stay true for at least 1 seconds
(abs(mag) < 0.1)
Fail
Pass
True
False
Untested
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34

Use the zoom, pan, and data cursor functionalities to analyze assessment evaluation results in the Expression Tree.

Link Temporal Assessments to Requirements

If you have a Simulink Requirements license, you can establish traceability between temporal assessments and requirements in Simulink Requirements by linking assessments to requirements. To create links to requirements, select the assessment in the editor and click the **Requirements** column to open the **Requirement Editor** dialog box. See “Link to Requirements” on page 1-2 for more information.

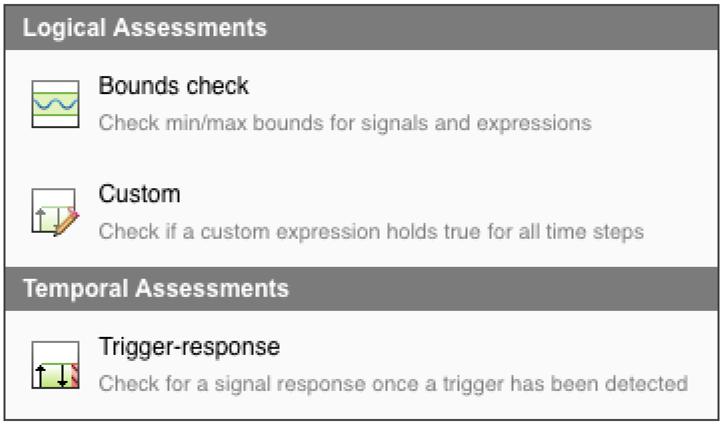
See Also

“Temporal Assessment Parameters” on page 3-88

Temporal Assessment Parameters

Simulink Test provides three temporal assessment templates:

- **Logical Assessment Templates**
  - **Bounds Check —** Check maximum and minimum bounds for signals and expressions.
  - **Custom —** Check if a logical expression holds true for all time steps.
- **Temporal Assessment Template**
  - **Trigger-Response —** Check for a signal response when a trigger is detected.



Logical Assessments

Bounds check

Check min/max bounds for signals and expressions

Custom

Check if a custom expression holds true for all time steps

Temporal Assessments

Trigger-response

Check for a signal response once a trigger has been detected

Bounds Check Assessments

Create bounds check assessments to check if the signals and expressions you test satisfy the boundary condition patterns you specify for them. Boundary condition pattern templates let you test if signals and expressions in terms of boundary values that you specify are:

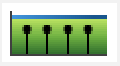
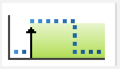
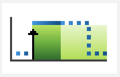
- **Always less than** (or equal to)
- **Always greater than** (or equal to)
- **Always inside**
- **Always outside**

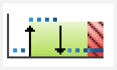
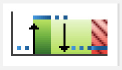
Trigger-Response Assessments

Create trigger-response assessments to verify a signal response when a trigger is detected. A trigger-response assessment requires:

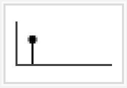
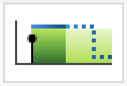
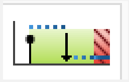
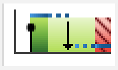
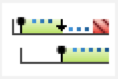
- `Trigger` parameter
- `Response` parameter
- Optional `Delay` parameter

The trigger condition is the condition pattern based on which the response signal is evaluated. There are five trigger condition patterns available:

Trigger Condition Pattern		Behavior	Available Time References
	**Whenever is true**	Check the response signal continuously whenever the triggering condition is true.	N/A
	**Becomes true**	Check the response signal every time the triggering condition becomes true.	Rising edge
	**Becomes true and stays true for at least**	Check the response signal every time the triggering condition becomes true and stays true for at least the interval specified by the `min-time` parameter (in s). You also specify an additional time reference parameter at which to evaluate the response signal.	Rising edge of trigger or end of `min-time`

Trigger Condition Pattern		Behavior	Available Time References
	**Becomes true and stays true for at most**	Check the response signal every time the triggering condition becomes true and stays true for at most the interval specified by the `max-time` parameter (in s). You also specify an additional time reference parameter at which to evaluate the response signal.	Rising or falling edge of trigger or end of `max-time`
	**Becomes true and stays true for between**	Check the response signal every time the triggering condition becomes true and stays true between the interval specified by the `min-time` and `max-time` parameters. You also specify an additional time reference parameter at which to evaluate the response signal.	Rising or falling edge of the trigger or end of `min-time` or `max-time`

To complete authoring a trigger-response assessment, you specify the response condition pattern and the response condition. There are five response condition patterns available:

Response Condition Pattern		Behavior
	**Must be true**	The response condition pattern must be true starting from the time reference parameter to the delay (if it is defined).
	**Must stay true for at least**	The response condition pattern must stay true for at least the duration specified by the `min-time` parameter.
	**Must stay true for at most**	The response condition pattern must stay true for at most the duration specified by the `max-time` parameter.
	**Must stay true for between**	The response condition pattern must stay true for at least the duration specified by the `min-time` parameter and at most the duration specified by the `max-time` parameter.
	**Must stay true until**	The response condition must stay true until the `until-condition` parameter becomes true within the duration specified by the `max-time` parameter.

The delay is an optional time interval starting from the time reference parameter to the point where the response condition is expected to be satisfied. You can set the delay to a maximum value or specify a time range in seconds.

Custom Assessments

The custom assessments template allows you to specify logical MATLAB expressions that do not fit in previous templates. Assessments are meant to evaluate signal properties, so all symbols defined in a custom template must be mapped to signal data (model element or timeseries or a constant scalar value).

See Also

"Assess Temporal Logic Using Temporal Assessments" on page 3-82

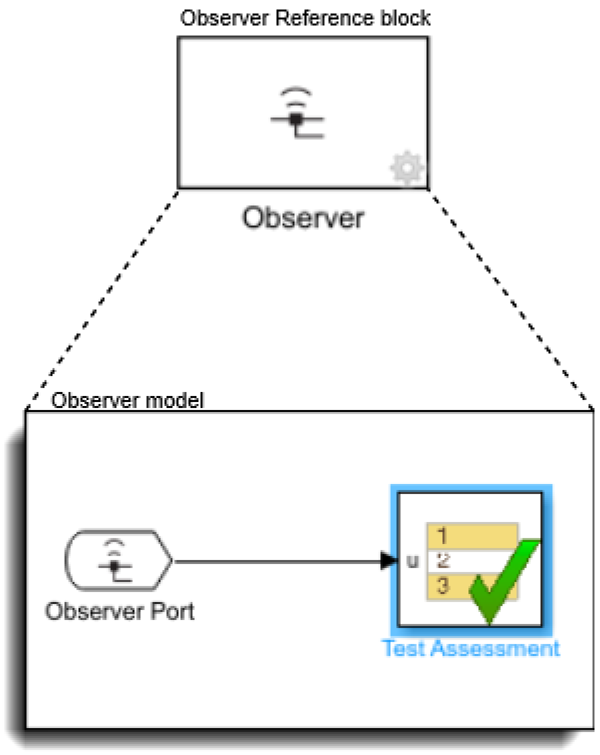
4

Observers

Access Model Data Wirelessly by Using Observers

In this section...

Observers allow you to you monitor the dynamic response of your system model while preserving the system model design and system result integrity. Observers are the Observer Reference block and the Observer Port block. The Observer Reference block wirelessly links a system model to an Observer model, which houses a verification subsystem. Inside an Observer model, you use Observer Port blocks to send signals from the system model to the verification subsystem.



Observer Reference Block

The Observer Reference block does not have inports or outports. You map your Simulinksignals to Observer Port blocks, which are contained within the Observer model. The Observer Port blocks are mapped to output data from your system model. Once the Observer Port blocks are mapped to a signal, you connect that signal line to the verification subsystem within the Observer model. Running your system model also causes the linked Observer model to run.

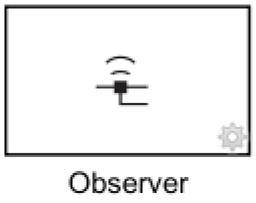
This wireless access allows you to use Observers to monitor your system model without causing changes to the system. Observers allow you to create a clear differentiation between your system design and verification subsystems.

For your Observer model to simulate, do not:

- Use a library model as an Observer model
- Include an Observer Reference block within an Observer model
- Reference the system model that contains the Observer Reference block from the Observer Reference block
- Use root inports within an Observer model
- Generate code from a system model that includes an Observer Reference model

Add an Observer Reference Block

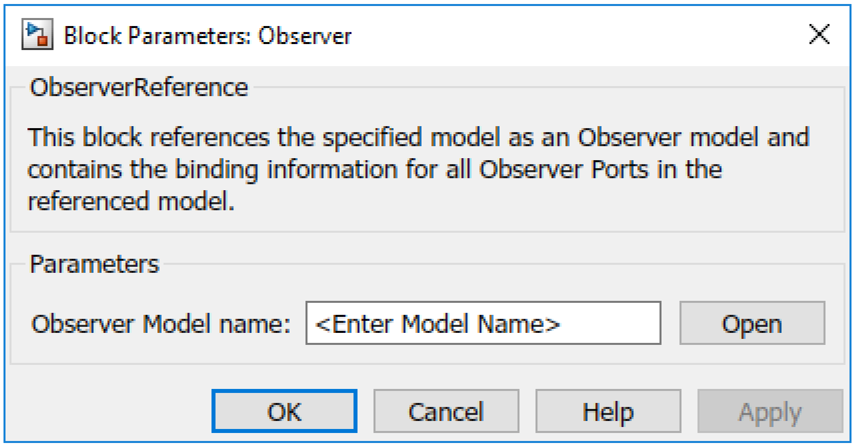
The Observer Reference block references a separate verification model that you use to verify your system model. To add an Observer Reference block to your system model, right-click the top level of your Simulink canvas. In the contextual menu, select **Observers > Add Observer here**.



An Observer Reference block is added to your system model, and an Observer model is created. You must save the Observer model in a directory on the MATLAB path.

Connect an Existing Observer Model

To connect an Observer Reference block to an Observer model that you have already created, first save your Observer model in a directory on the MATLAB path. Next, open the Observer Reference block parameters by right-clicking the Observer Reference block. Select **Block Parameters (ObserverReference)**.


Block Parameters: Observer

ObserverReference

This block references the specified model as an Observer model and contains the binding information for all Observer Ports in the referenced model.

Parameters

Observer Model name: <Enter Model Name>

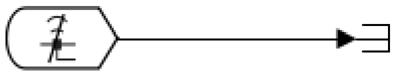
Enter the name of the Observer model that you want to connect to your system and select **Apply**. When you double click your Observer Reference block, your Observer model opens in a new window.

Create an Observer Model from Signal Lines

To create an Observer model that is mapped to a signal line in your model, select and right-click on one or more signals that you want to verify. Select **Observers** > **Observe Selected Signals** > **New Observer**. Simulink creates a new Observer model and adds an Observer Reference block to your system model.

Connect Signals by Using an Observer Port Block

Each Observer Reference block contains one or more Observer Port blocks. After mapping an Observer Port block to an object or signal within a system model, the Observer Port block outputs the same output as its mapped object.



A new Observer Port block shows a line through the signal symbol, signifying that the block is not mapped to a signal.

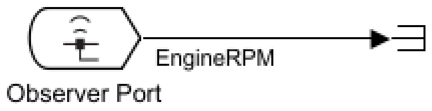
Observer Dialog Box

To map an Observer Port block to a signal on your system model, select **Analysis** > **Observers** > **Manage Observer...**. This opens the Observer Dialog box. Within the Observer Dialog box you can:

- Filter and select signals for observation
- Add, remove, or configure Observer Port blocks
- Trace signals

On the left-hand side of the Observer Dialog box is the Observable Area pane. The Observable Area pane displays the block hierarchy and observable outputs of your model. Observed signals appear bold in the hierarchy.

The right hand side of the Observer Dialog box shows the Observer pane. The Observer pane displays the block hierarchy within the Observer Reference block. An Observer Port block that is mapped to a signal appears bold and displays the signal to which it is attached. Once the Observer Port is mapped to a signal, the symbol updates to show that the Observer Port is attached to a signal.



To view the full path of an observed object, hover your cursor over the Observer Port block.

If you change the name of an observed signal in your system model, the Observer Reference block updates the name of the output signal from the Observer Port block. If a signal is not named and does not have a label, the output of the Observer Port block is set to an empty string.

Map an Observer Port Block to a Signal

To map a signal to an Observer Port block, open the Observer Dialog box. In the Observable Area pane, select the signal that you want to observe. You can map the signal to an existing Observer Port block by redirect Observer Port icon. You can also map the signal to a new Observer Port block by selecting the add new Observer Port icon. You can then connect the signal to a verification subsystem to test your results.

Simulate a System Model with an Observer Reference Block

The Observer model is used to monitor signals in your system model and checks that your system model is running within specified parameters. With or without an Observer Reference block, your system model simulation results are the same. The Observer Reference block does not affect the compilation of your system model and supports only normal simulation mode. The Observer Reference block supports multiple execution rates, continuous dynamics, and zero crossings.

Before running a system model that includes an Observer Reference block, configure your system model and your Observer model to use a fixed-step solver. See “Choose a Fixed-Step Solver” (Simulink). Set the simulation mode for both to normal. These signal types are supported:

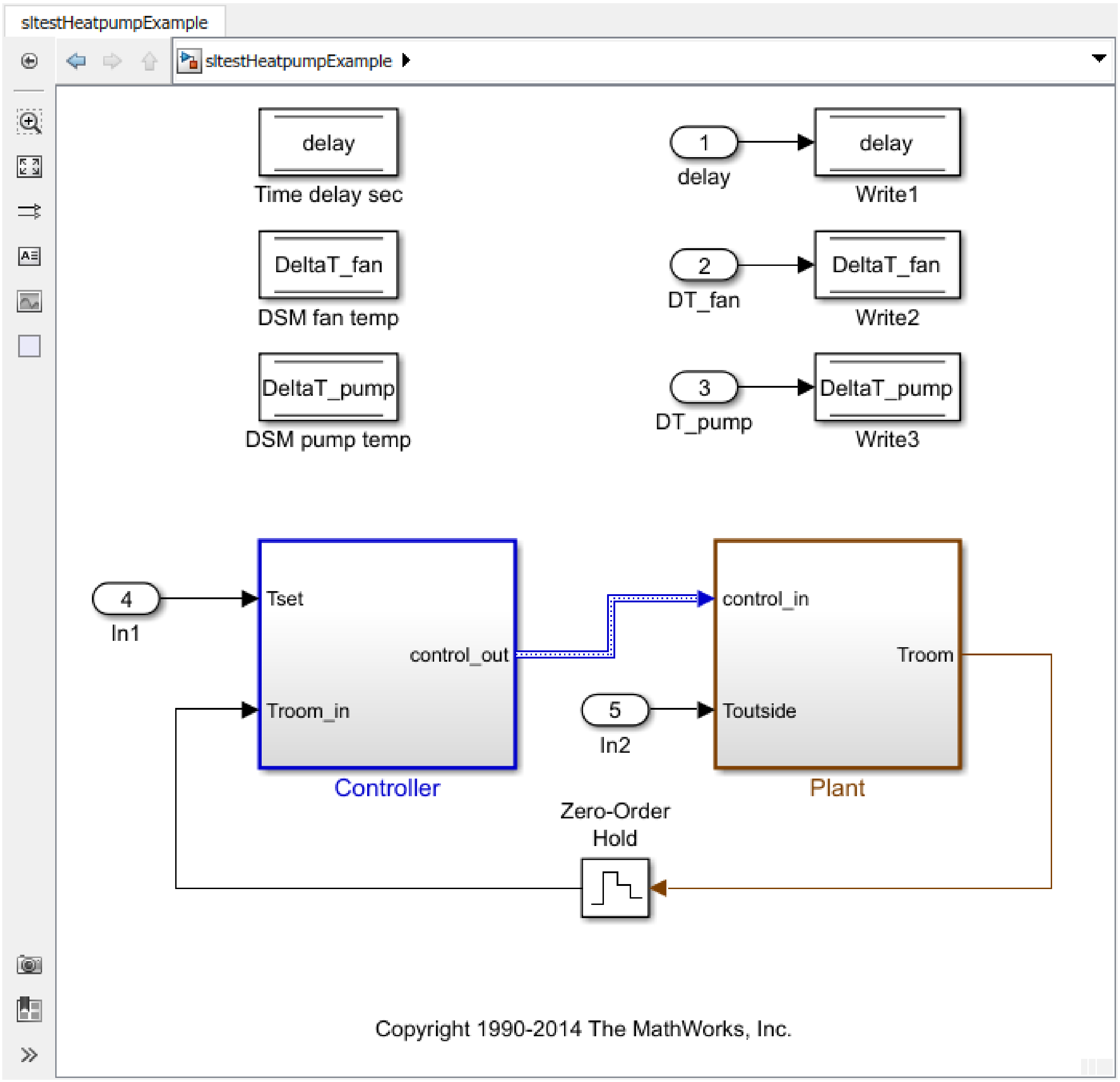
- Scalar
- Wide
- Nonvirtual buses
- Continuous
- Zero-Order Hold
- Discrete

Verify Heat Pump Temperature by Using Observers

This example shows how to use an Observer Reference block to wirelessly observe signals and verify results. In this system, the plant is modeled using Simulink, and the controller is modeled using Stateflow. The goal of the example is to monitor the temperature of the heat pump as well as when the pump is cooling or heating the room. Cooling and heating are denoted by which direction the fan is blowing. The data name is `pump_dir`, and it is connected to port 3 in the Stateflow chart.

To open this example, enter:

```
cd(fullfile(docroot,'toolbox','sltest','examples'))
open_system('sltestHeatpumpExample')
```

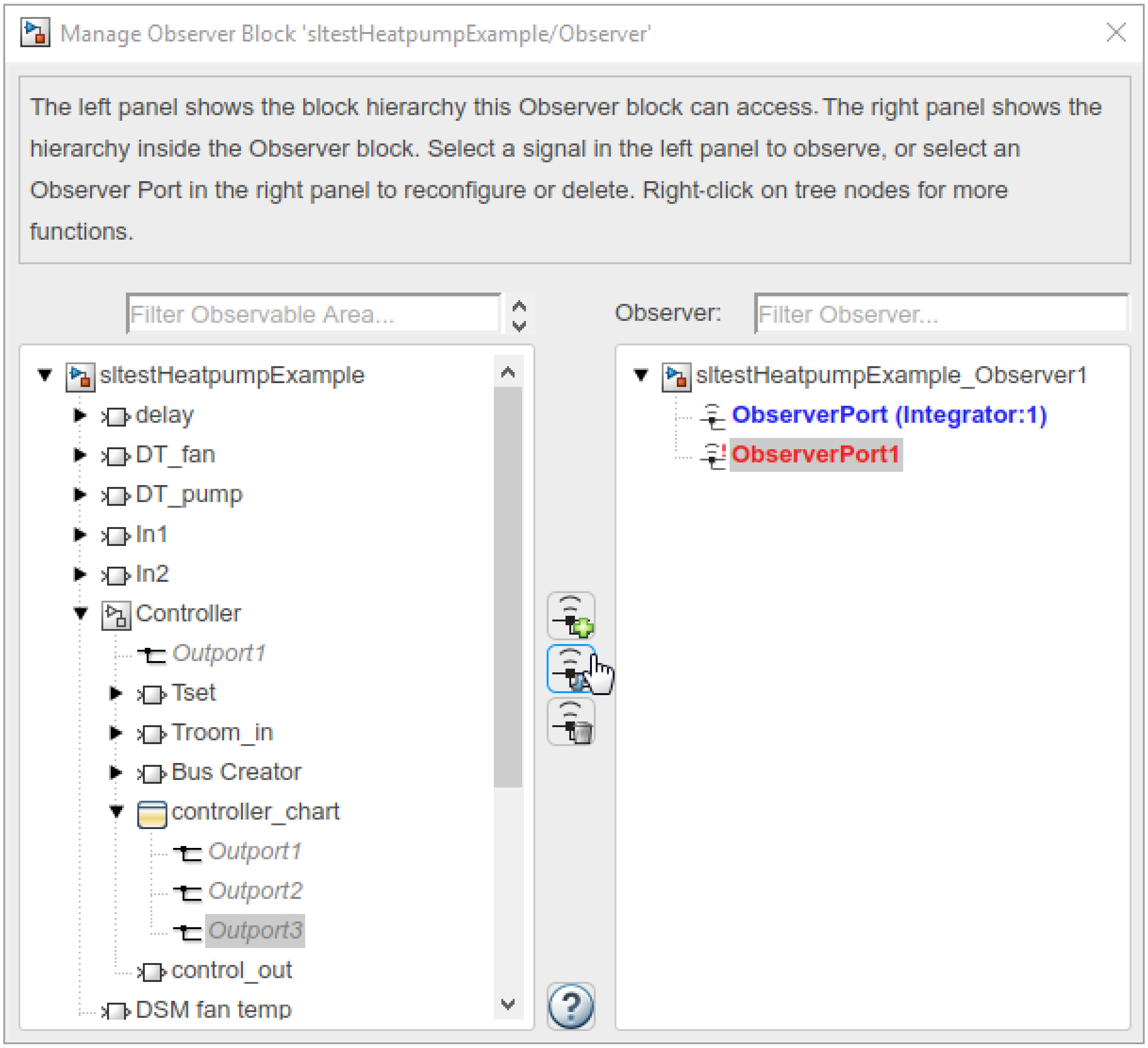


To create a new Observer model to measure the temperature of the pump, open the plant model and highlight the signal T. Right-click the hightlighted signal and select **Observers**

> **Observe selected signals** > **New Observer**. Simulink adds an Observer Reference block to your system model and creates a new Observer model called `sltestHeatpumpExample_Observer1`. The Observer model contains an Observer Port block that is mapped to the signal `T`. Save the new Observer model in the same directory that contains the heat pump model.

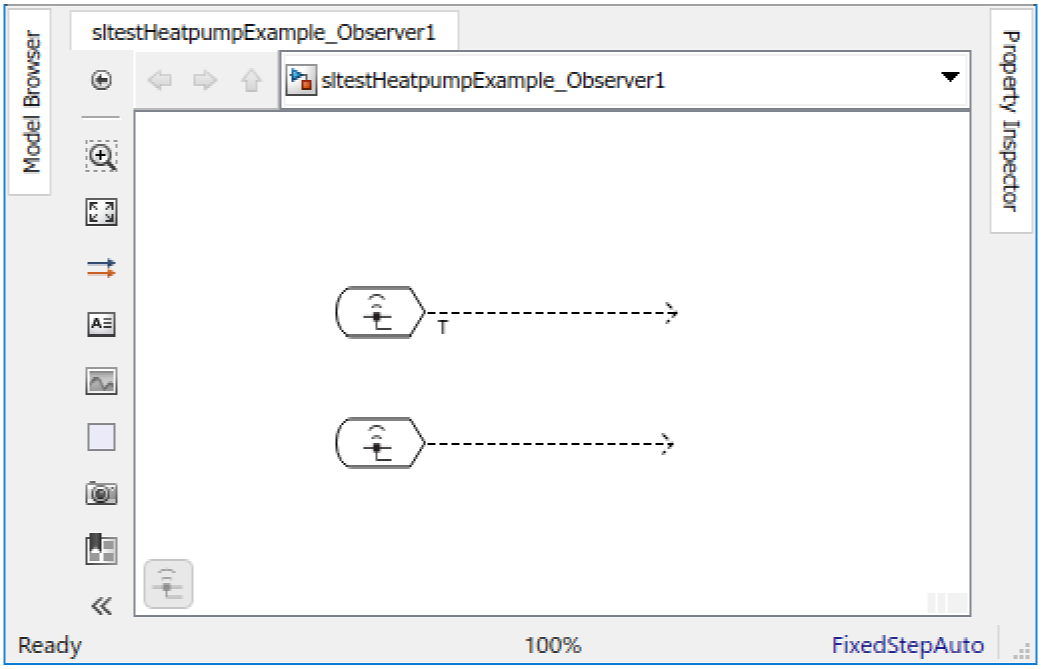
Add a second Observer Port block to your Observer model. Double-click the Observer Port to open the Observer Dialog. In the Observer pane, the second Observer Port, `ObserverPort1`, is listed below the first port.

To map the second `ObserverPort1` to the Simulink data `pump_dir`, click on `ObserverPort1` and `Outport3`. Once both are highlighted, click the Reconfigure button .


Manage Observer Block 'sltestHeatpumpExample/Observer'

The left panel shows the block hierarchy this Observer block can access. The right panel shows the hierarchy inside the Observer block. Select a signal in the left panel to observe, or select an Observer Port in the right panel to reconfigure or delete. Right-click on tree nodes for more functions.

The two Observer Port blocks are now both mapped to the signals and are ready to be connected to scopes or a verification subsystem.

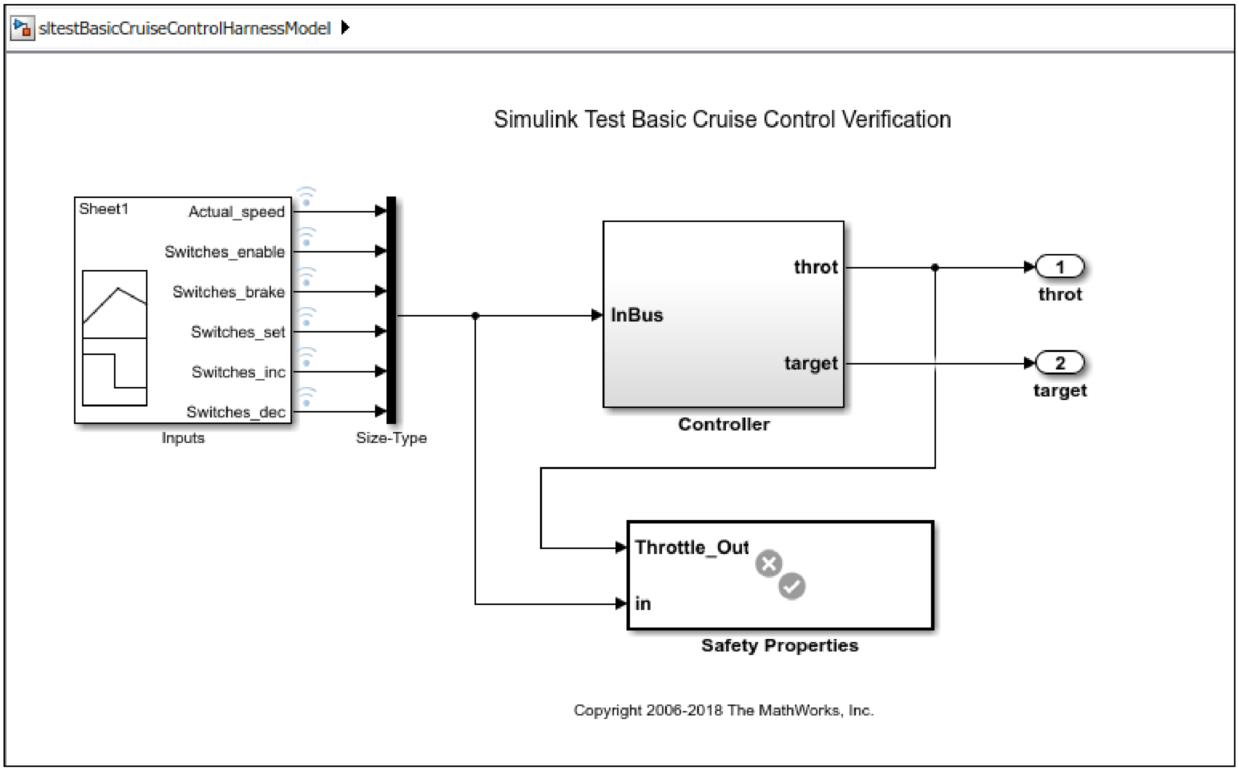

sltestHeatpumpExample_Observer1
Model Browser
Property Inspector
sltestHeatpumpExample_Observer1
T
Ready
100%
FixedStepAuto

Convert Verification Subsystem to an Observer Reference

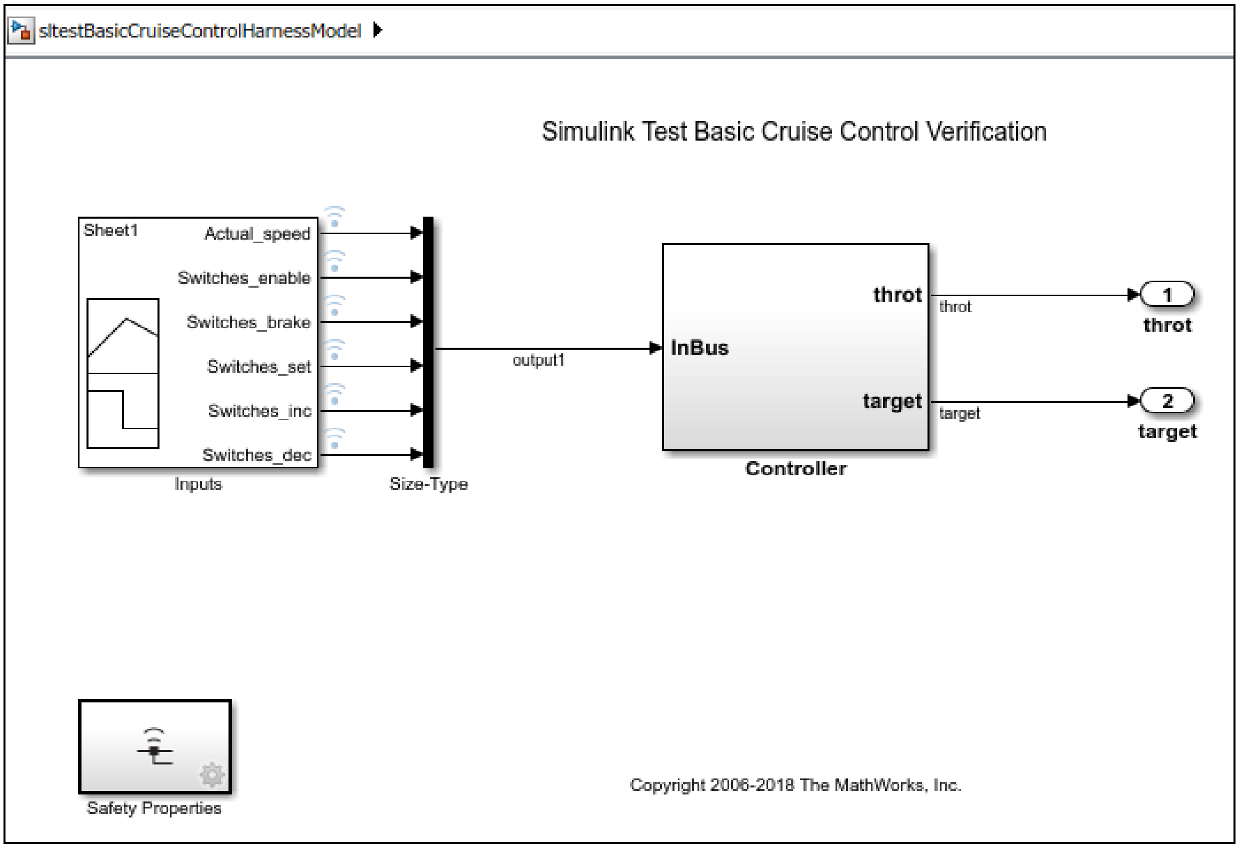
To convert a verification subsystem to an Observer Reference block, right-click the verification subsystem. Select **Observers** > **Move selected block to Observer** > **New Observer**. This operation can not be undone.

Declutter a System Model by Using an Observer Reference Block

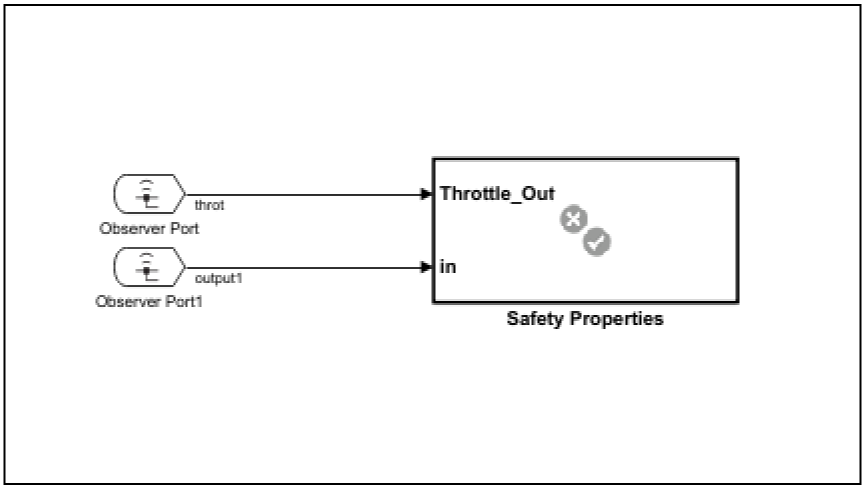
In this example, a cruise control system generates the trottle and the target speed. The Safety Properties block is a verification subsystem that verifies the safety of the cruise control system.



By converting the verification subsystem to an Observer Reference block, you remove the signals that link the verification subsystem to the system model while preserving the ability to test the integrity of the system.



The two signals, `throt` and `output1`, are automatically mapped to two Observer Port blocks within the Observer model, `sltestBasicCruiseControlHarnessModel_Observer1`.



See Also

More About

- Observer Port
- Observer Reference

Test Harness Software- and Processor-in-the-Loop

SIL Verification for a Subsystem

In this section...

This example shows subsystem verification by ensuring the output of software-in-the-loop (SIL) code matches that of the model subsystem. You generate a SIL verification harness, collect simulation results, and compare the results using the simulation data inspector. You can apply a similar process for processor-in-the-loop (PIL) verification.

With SIL simulation, you can verify the behavior of production source code on your host computer. Additionally, with PIL simulation, you can verify the compiled object code that you intend to deploy in production. You can run the PIL object code on real target hardware or on an instruction set simulator.

If you have an Embedded Coder license, you can create a test harness in SIL or PIL mode for model verification. You can compare the SIL or PIL block results with the model results and collect metrics, including execution time and code coverage. Using the test harness to perform SIL and PIL verification, you can:

- Manage the harness with your model. Generating the test harness generates the SIL block. The test harness is associated with the component under verification. You can save the test harness with the main model.
- Use built-in tools for these test-design-test workflows:
  - Checking the SIL or PIL block equivalence
  - Updating the SIL or PIL block to the latest model design
- View and compare logged data and signals using the Test Manager and Simulation Data Inspector.

This example models a closed-loop controller-plant system. The controller regulates the plant output.

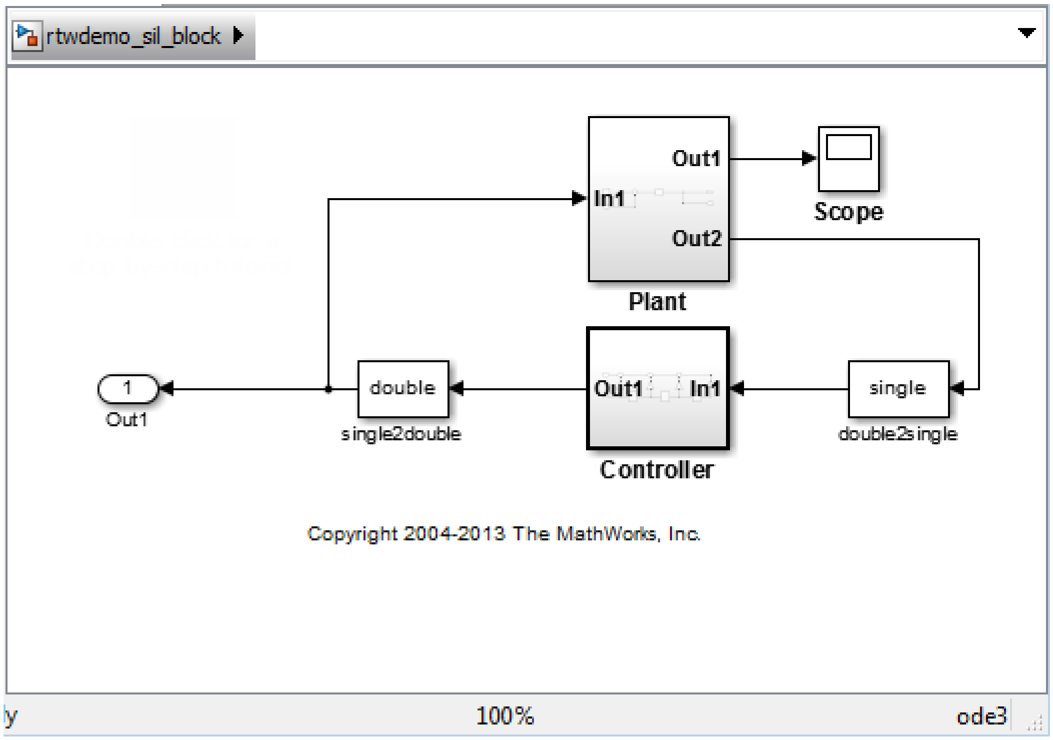
Create a SIL Verification Harness for a Controller

Create a SIL verification harness using data that you log from a controller subsystem model simulation. You need an Embedded Coder license for this example.

1 Open the example model by entering

```
rtwdemo_sil_block
```

at the MATLAB command prompt,



2 Save a copy of the model using the name `controller_model` in a new folder, in a writable location on the MATLAB path.

3 Enable signal logging for the model. At the command prompt, enter

```
set_param(bdroot,'SignalLogging','on','SignalLoggingName',...
'SIL_signals','SignalLoggingSaveFormat','Dataset')
```

4 Right-click the signal into Controller port In1, and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `controller_model_input`. Select **Log signal data** and click **OK**.

5 Right-click the signal out of Controller port Out1, and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `controller_model_output`. Select **Log signal data** and click **OK**.

6 Simulate the model.

7 Get the logged signals from the simulation output into the workspace. At the command prompt, enter

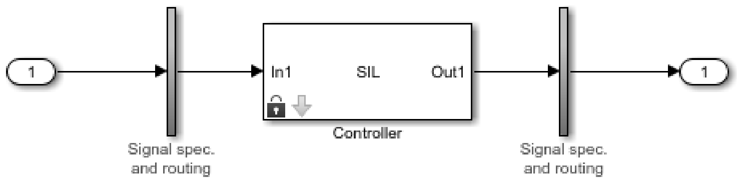
```
out_data = out.get('SIL_signals');
control_in1 = out_data.get('controller_model_input');
control_out1 = out_data.get('controller_model_output');
```

8 Create the software-in-the-loop test harness. Right-click the Controller subsystem and select **Test Harness > Create Test Harness (Controller)**.

9 Set the harness properties:

- **Name**: `SIL_harness`
- **Sources and Sinks**: `Inport` and `Outport`
- Select **Open harness after creation**
- **Advanced Properties – Verification Mode**: `Software-in-the-loop (SIL)`

Click **OK**. The resulting test harness has a SIL block.



Configure and Simulate a SIL Verification Harness

Configure and simulate a SIL verification harness for a controller subsystem.

1 Configure the test harness to import the logged controller input values. From the top level of the test harness, in the model **Configuration Parameters** dialog box, in the

Data Import/Export pane, select **Input**. Enter `control_in1.Values` as the input and click **OK**.

2 Enable signal logging for the test harness. At the command prompt, enter

```
set_param('SIL_harness','SignalLogging','on','SignalLoggingName',...
'harness_signals','SignalLoggingSaveFormat','Dataset')
```

3 Right-click the output signal of the SIL block and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `SIL_block_out`. Select **Log signal data** and click **OK**.

4 Simulate the harness.

Compare the SIL Block and Model Controller Outputs

Compare the outputs for a verification harness and a controller subsystem.

1 In the test harness model, click the Simulation Data Inspector button to open the Simulation Data Inspector.

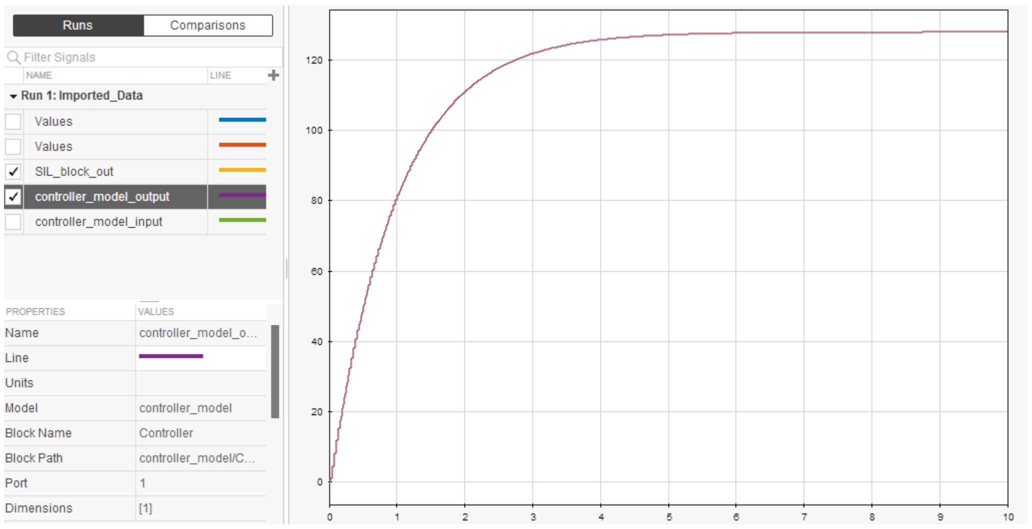
2 In the Simulation Data Inspector, click **Import**. In the **Import** dialog box.

- Set **Import from** to: `Base workspace`.
- Set **Import to** to: `New Run`.
- Under **Data to import**, select **Signal Name** to import data from all sources.

3 Click **Import**.

4 Select the `SIL_block_out` and `controller_model_out` signals in the **Runs** pane of the data inspector window.

The chart displays the two signals, which overlap. This result suggests equivalence for the SIL code. You can plot signal differences using the **Compare** tab in SDI, and perform more detailed analyses for verification. For more information, see “Compare Simulation Data” (Simulink).



	NAME	LINE
☐	Values	🟦
☐	Values	🟧
✓	SIL_block_out	🟨
✓	controller_model_output	🟪
☐	controller_model_input	🟩

PROPERTIES	VALUES
Name	controller_model_o...
Line	🟪
Units	
Model	controller_model
Block Name	Controller
Block Path	controller_model/C...
Port	1
Dimensions	[1]

5 Close the test harness window. You return to the main model. The badge ▥ on the Controller block indicates that the SIL harness is associated with the subsystem.

See Also

More About

- “Control Generation of Functions for Subsystems” (Simulink Coder)
- “Configure and Run SIL Simulation” (Embedded Coder)

Test Integrated Code

In this section...

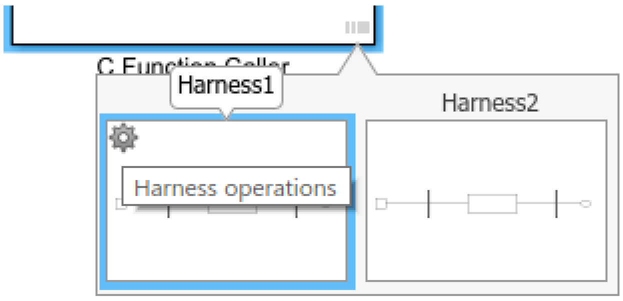
Test Integrated C Code

If you have a model that integrates C code with a C Caller block, you can test the C code with the Test Manager and a test harness. For an example, see "C Code Verification with Simulink Test".

The C Caller block uses configuration parameters to define the custom code. If you change the configuration parameters, synchronize the parameters between the test harness and the model. For more information, see "Synchronize Changes Between Test Harness and Model" on page 2-53 and "Create Test Harnesses and Select Properties" on page 2-13.

- If you change the test harness configuration parameters, you can push the configuration set to the main model. Select **Analysis > Test Harness > Push Component and Parameters to Main Model**, or use `sltest.harness.push`.
- If you change the main model configuration parameters in the main model, and you want to update the test harness parameters, the test harness must copy the configuration parameters on rebuild. You can set this property in two ways:

  - When you create the test harness, in the **Advanced Properties**, select **Update Configuration Parameters and Model Workspace data on rebuild**.
  - For existing test harnesses, in the harness preview, click the **Harness operations** icon to open the harness properties. In the harness properties, select **Update Configuration Parameters and Model Workspace data on rebuild**.



Test Code in S-Functions

S-Functions are computer language descriptions of Simulink blocks written in MATLAB, C, C++ or Fortran. You can test code wrapped in S-Functions using Simulink Test test harnesses. Testing code in S-Functions can be helpful for regression testing of legacy code and for testing your code in a system context.

S-Function Testing Example

In this example, you test code in an S-Function block using a test harness. The main model is a controller-plant model of an air conditioning/heat pump unit. Before you begin, change the default working folder to one with write permissions.

Note This example works only on a 64–bit Windows® platform.

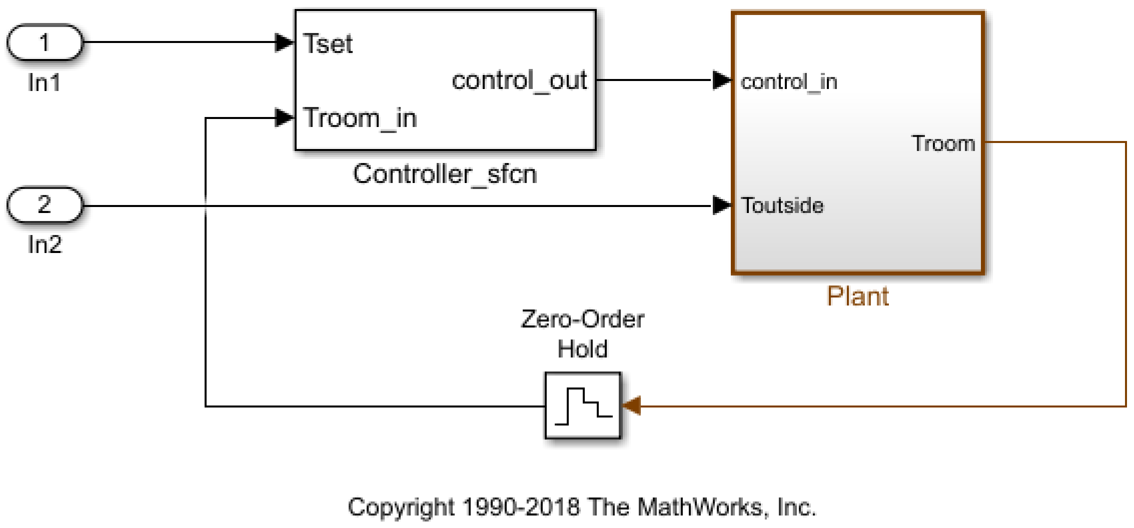
Set Up the Working Environment

1 Add the example folder to the MATLAB path, and set the example file names.

```
ep = fullfile(docroot,'toolbox','sltest','examples');
addpath(ep);

md = 'sltestHeatpumpSfunExample.slx'
cb = 'sltestHeatpumpBusPostLoadFcn.mat'
dt = 'PumpDirection.m'
```

2 Open the model.

```
open_system(fullfile(ep,md))
```



In the example model:

- The controller is an S-Function that accepts room temperature and specified temperature inputs.
- The controller output is a bus with signals that control the fan, heat pump, and the direction of the heat pump (heat or cool).
- The plant accepts the control bus. The heat pump and the fan signals are Boolean, and the heat pump direction is specified by `+1` for cooling and `-1` for heating.

The test covers four temperature conditions. Each condition corresponds to one operating state with fan, pump, and pump direction signal outputs.

Temperature Condition	**System State**	**Fan Command**	**Pump Command**	**Pump Direction**
`|Troom_in - Tset| < DeltaT_fan`	idle	0	0	0
`DeltaT_fan <= |Troom_in - Tset| < DeltaT_pump`	fan only	1	0	0

Temperature Condition	System State	Fan Command	Pump Command	Pump Direction
`|Troom_in - Tset| >= DeltaT_pump and Tset < Troom_in`	cooling	1	1	-1
`|Troom_in - Tset| >= DeltaT_pump and Tset > Troom_in`	heating	1	1	1

Create a Test Case

1. Open the Test Manager by selecting **Analysis** > **Test Manager** from the Simulink menu.
2. From the Test Manager toolstrip, click **New** to create a test file. Name and save the test file.
3. In the test case, under **System Under Test** , click the button to load the current model into the test case.

Create a Test Harness

1. In the model, right-click the `Controller_sfcn` subsystem and select **Test Harness > Create for 'Controller_sfcn'**.
2. Set the harness properties.

   In the **Basic Properties** tab:

   - Set **Name** to `test_harness_1`
   - Set **Sources and Sinks** to **None** and **Scope**
3. Click **OK** to create the test harness.
4. In the test case, under **System Under Test**, refresh the test harness list and select `test_harness_1` for the **Harness**.

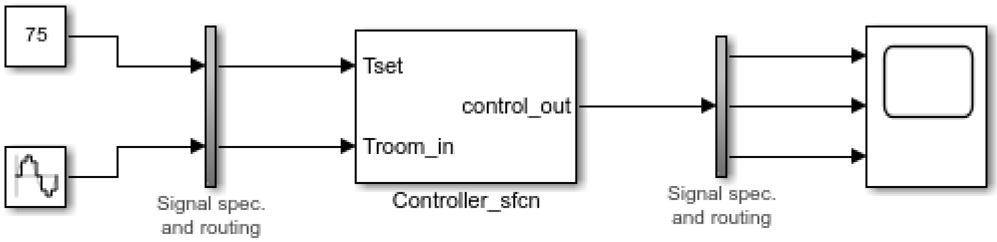
Add Inputs and Set Simulation Parameters

Create inputs in the test harness, with a constant `Tset` and a time-varying `Troom_in`.

1. Connect a Constant block to the `Tset` input and set the value to `75`.
2. Add a Sine Wave block to the harness model to simulate a temperature signal. Connect the Sine Wave block to the conversion subsystem input `Troom_in`.
3. Double-click the Sine Wave block and set the parameters:

Parameter	Value
Amplitude	15
Bias	75
Frequency	2*pi/3600
Phase (rad)	0
Sample time	1

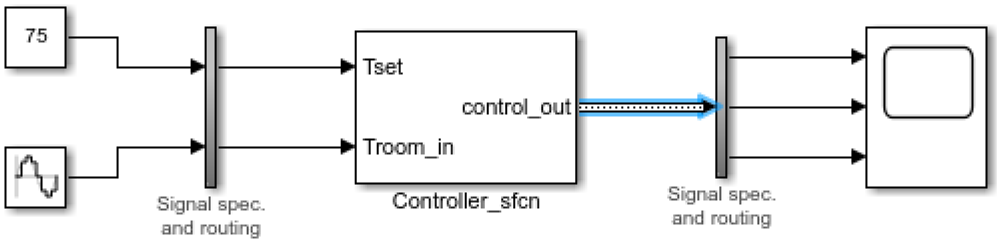
Select **Interpret vector parameters as 1-D**.



4 In the **Solver** pane of the Simulink toolstrip, set **Stop time** to `3600`.

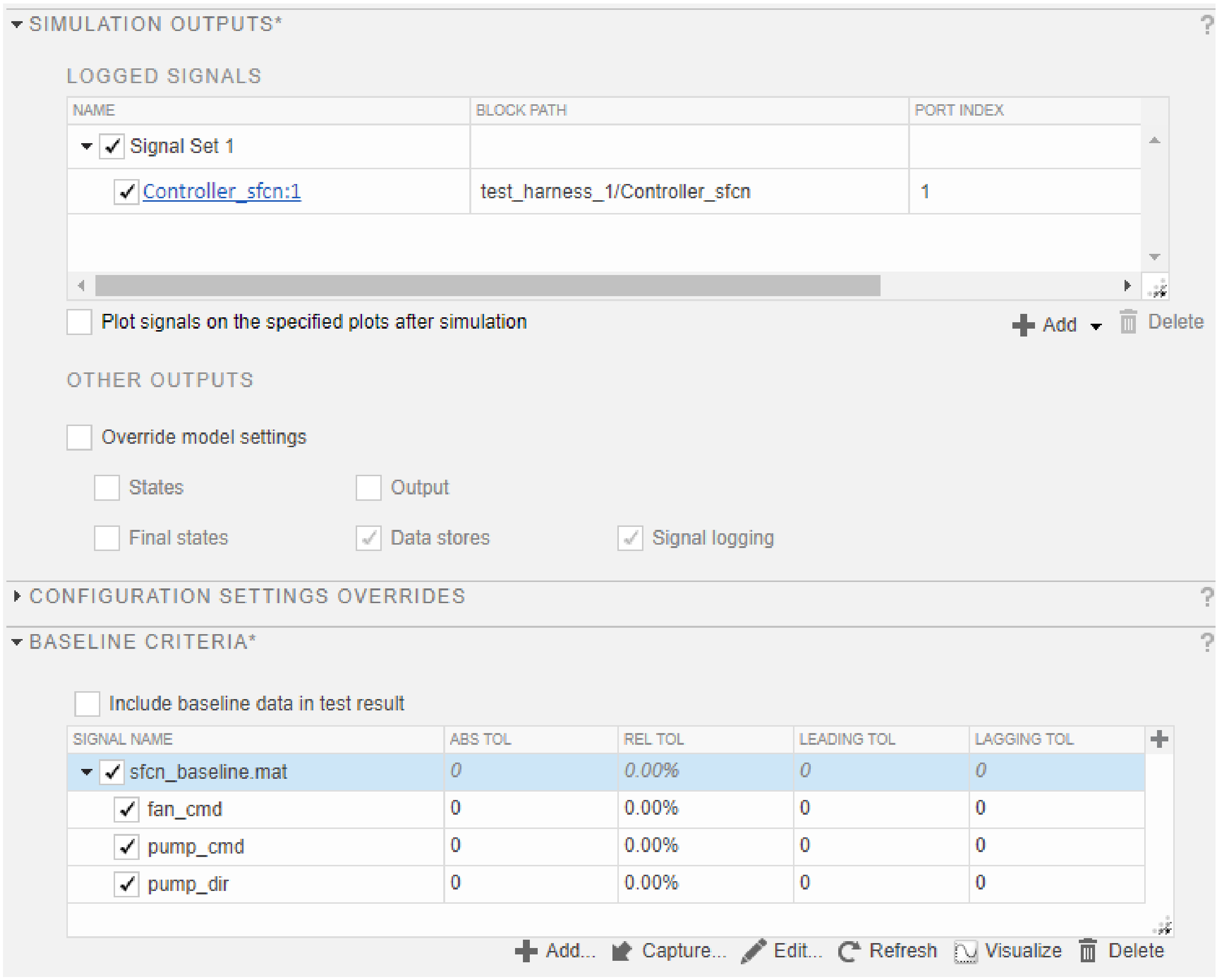
Obtain Baseline Data

1 In the test case, in **Simulation Outputs**, click **Add**. Highlight the output bus from the controller S-Function.



2 In the **Signal Selection** dialog box, click the **Add** button.

3 Under **Baseline Criteria**, click **Capture** to record a baseline data set from simulating the test harness. Save the baseline data set to the working folder. The baseline signals appear in the table.


SIMULATION OUTPUTS*

LOGGED SIGNALS

NAME	BLOCK PATH	PORT INDEX
✓ Signal Set 1		
✓ Controller_sfcn:1	test_harness_1/Controller_sfcn	1

☐ Plot signals on the specified plots after simulation

OTHER OUTPUTS

☐ Override model settings

☐ States ☐ Output

☐ Final states ✓ Data stores ✓ Signal logging

CONFIGURATION SETTINGS OVERRIDES

BASELINE CRITERIA*

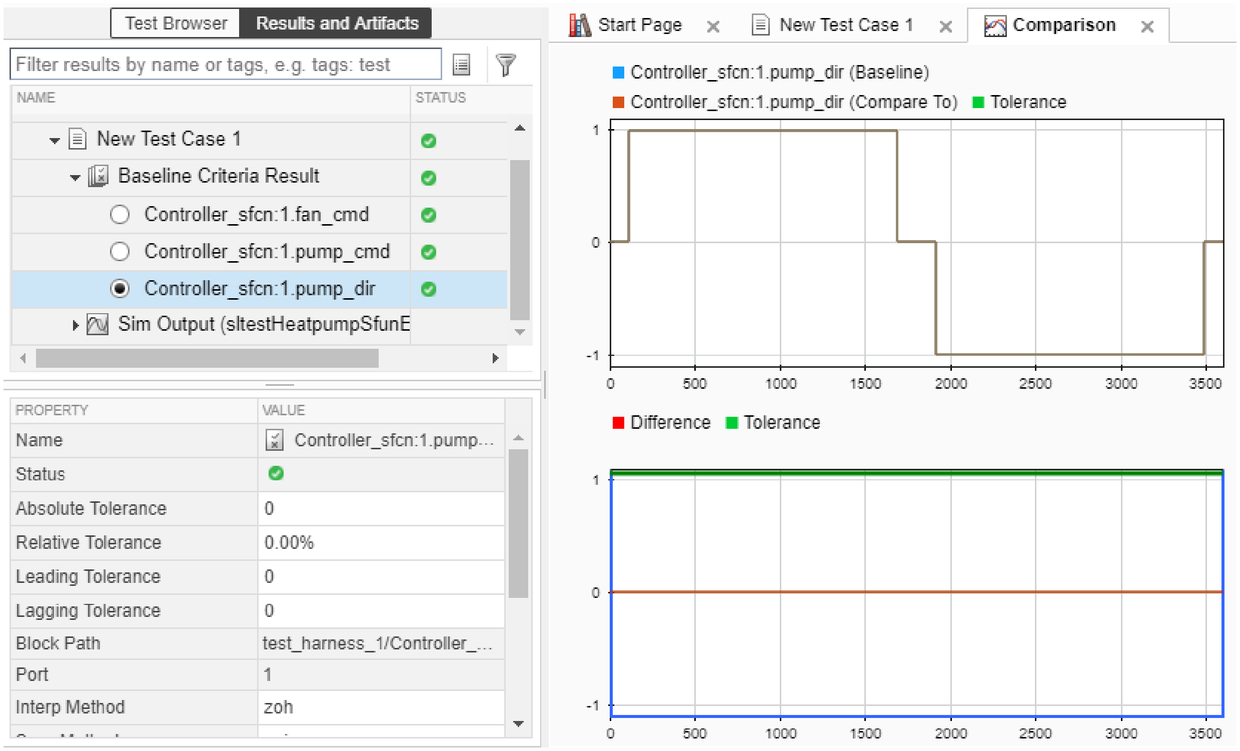
☐ Include baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
✓ sfcn_baseline.mat	0	0.00%	0	0
✓ fan_cmd	0	0.00%	0	0
✓ pump_cmd	0	0.00%	0	0
✓ pump_dir	0	0.00%	0	0

Run the Test Case and View Results

1 Run the test case. The test results appear in the **Results and Artifacts** pane.

2 Expand the results to view the baseline criteria result. The baseline test passes because the simulation output is identical to the baseline data.


Test Browser
Results and Artifacts
Filter results by name or tags, e.g. tags: test
Start Page
New Test Case 1
Comparison
Controller_sfcn:1.pump_dir (Baseline)
Controller_sfcn:1.pump_dir (Compare To)
Tolerance
Difference
Tolerance

NAME	STATUS
New Test Case 1	✓
Baseline Criteria Result	✓
Controller_sfcn:1.fan_cmd	✓
Controller_sfcn:1.pump_cmd	✓
● Controller_sfcn:1.pump_dir	✓
Sim Output (sltestHeatpumpSfunE	

PROPERTY	VALUE
Name	Controller_sfcn:1.pump…
Status	✓
Absolute Tolerance	0
Relative Tolerance	0.00%
Leading Tolerance	0
Lagging Tolerance	0
Block Path	test_harness_1/Controller_…
Port	1
Interp Method	zoh

Simulink Test Manager Introduction

Functional Testing for Verification

In this section...

You can use Simulink Test to author, manage, and execute tests for Simulink models and generated code. You can author tests from scratch, import existing test data and harness models, and organize tests using the Test Manager. You can execute tests in model, software-in-the-loop (SIL), processor-in-the-loop (PIL), and hardware-in-the-loop (HIL) modes, control parameters, and iterate over parametric values. You can run test cases individually, in batch, or as a filtered subset of the test file. You can also run the same tests back-to-back in multiple releases of MATLAB.

Results include a concise pass/fail summary for elements in your test hierarchy, including iterations, test cases, test suites, and the test file. Visualization tools help you drill down into individual data sets to determine, for example, the time and cause of a particular failure. Coverage results from Simulink Coverage help quantify the extent to which your model or code is tested.

For example, you can:

- Compare results between your model and generated code by running back-to-back equivalence tests between different environments, such as model simulation, SIL, PIL, and HIL execution.
- Optimize your model or code by iterating over parametric values or configuration parameters.
- Start testing on a unit level by using test harnesses, and reuse those tests as you scale up to the integration and system level.
- Run models that contain test vectors and assessments inside the Simulink block diagram.

Simulink Test includes a comprehensive programmatic interface for writing test scripts, and Simulink tests can be integrated with MATLAB tests using MATLAB Unit Test.

Test Authoring

When you author a test, you define test inputs, signals of interest, signal pass/fail tolerances, iterations over parametric values, and assessments for simulation behavior. You can author test input vectors in several ways:

- Graphically, such as with the Signal Editor
- From datasets, such as using Excel® or MAT files
- As a sequence of test steps that progresses according to time or logical conditions

You can define assessments to indicate when functional requirements are not met. These assessments follow from your design requirements or your test plan. You can define assessments in several ways:

- With a structured assessment language. The structured language helps you assess complex timing behavior, such as two events that must happen within a certain time frame. It also helps you identify conflicts between requirements.
- With `verify` statements in a Test Assessment or Test Sequence block. For information on how to set up the blocks in your model, see "Assess Model Simulation Using verify Statements" on page 3-15.
- With blocks in the Model Verification block library.
- With tolerances you set on the simulation data output. Tolerances define the acceptable delta from baseline data or another simulation.
- With a custom criteria script that you author using MATLAB.

You can use existing test data and test models with Simulink Test. For example, if you have data from field testing, you can test your model or code by mapping the data to your test case. If you have existing test models that use Model Verification blocks, you can organize those tests and manage results in the Test Manager.

Test Generation

Using Simulink Design Verifier, you can generate test cases that achieve test objectives or increase model or code coverage. You can generate test cases from the Test Manager, or from the Simulink Design Verifier interface. Either way, you can include the generated test cases with your original tests to create a test file that achieves complete coverage. You can also link the new test cases to additional requirements.

Test Execution

You can control test execution modes from the Test Manager. For example, you can:

- Run tests in multiple releases of MATLAB. Multiple release testing allows you to leverage recent test data while executing your model in its production version.
- Run back-to-back tests to verify generated code. You can run the same test in model, SIL, and PIL mode and compare numeric results to demonstrate code-model equivalence.
- Run HIL tests to verify systems running on real-time hardware using Simulink Real-Time™, including `verify` statements in your model that help you determine whether functional requirements are met.
- Decrease test time by running tests in parallel using the Parallel Computing Toolbox™, or running a filtered subset of your entire test file.

Reporting

When reporting your test results, you can set report properties that match your development environments. For example, reporting can depend on whether tests passed or failed, and reports can include data plots, coverage results, and requirements linked to your test cases. You can create and store custom MATLAB figures that render with a report. Reporting options persist with your test file, so they run every time you execute a test.

A MATLAB Report Generator™ license adds additional customization options, including:

- Creating reports from a Microsoft® Word or PDF template
- Assembling reports using custom objects that aggregate individual results

See Also

Related Examples

- “Create and Run a Baseline Test”
- “Inputs”
- “Test Execution”

Test Manager Test Cases

Manage Test File Dependencies

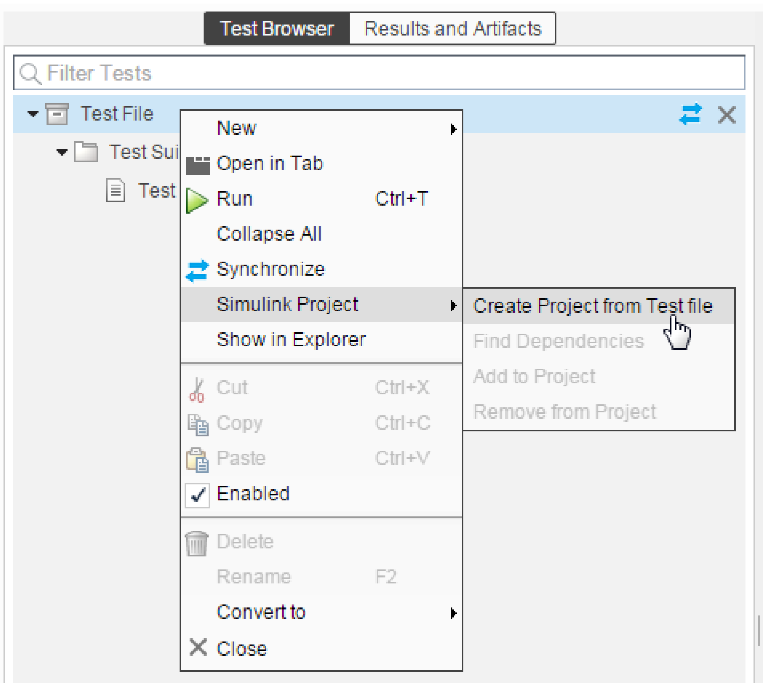
In this section...

You can help track and manage your test file dependencies by creating a Simulink Project for your test file and the files it depends on. Examples of test file dependencies include requirements, data files, callbacks, test harnesses, and custom criteria scripts. Packaging test file dependencies in a Simulink Project also helps you share tests with other users.

Package a Test File Using Simulink Projects

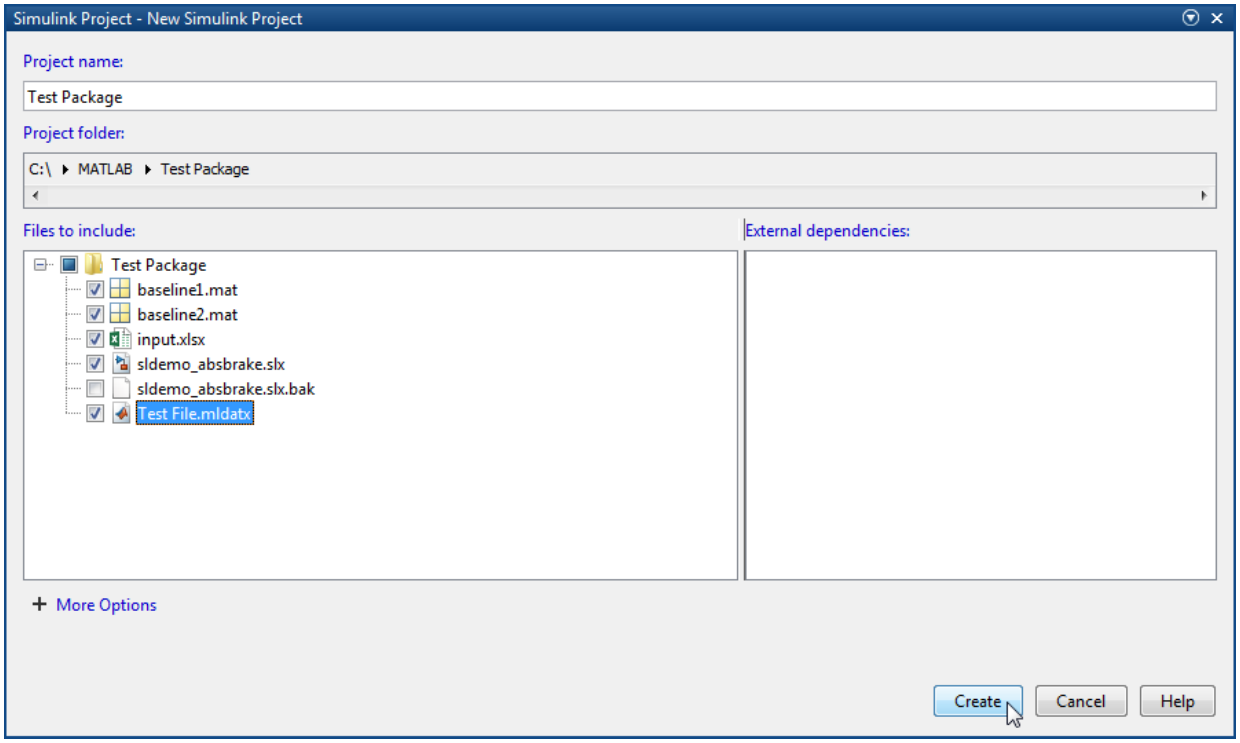
1. In the **Test Browser**, right-click the test file.
2. Select **Simulink Project > Create Project from Test File**.

   Simulink Projects opens and identifies the file dependencies of the test file. In this example, the test file contains a test case with a requirements link, an input file, and a baseline file.



3 Specify project name, and verify the list of selected file dependencies.

4 Click **Create**.


Simulink Project - New Simulink Project

Project name:

Test Package

Project folder:

C:\ ▸ MATLAB ▸ Test Package

Files to include:

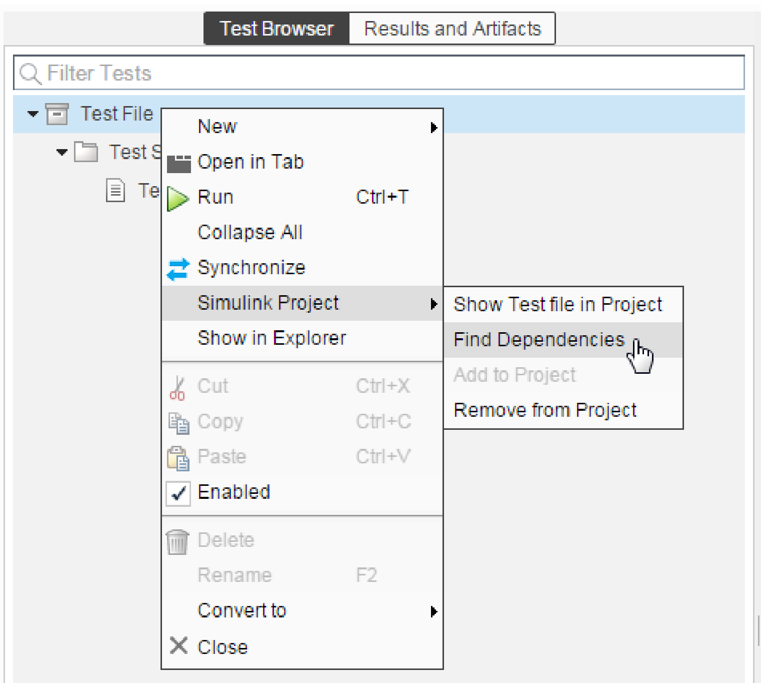
- Test Package
  - [✓] baseline1.mat
  - [✓] baseline2.mat
  - [✓] input.xlsx
  - [✓] sldemo_absbrake.slx
  - [ ] sldemo_absbrake.slx.bak
  - [✓] Test File.mldatx

External dependencies:

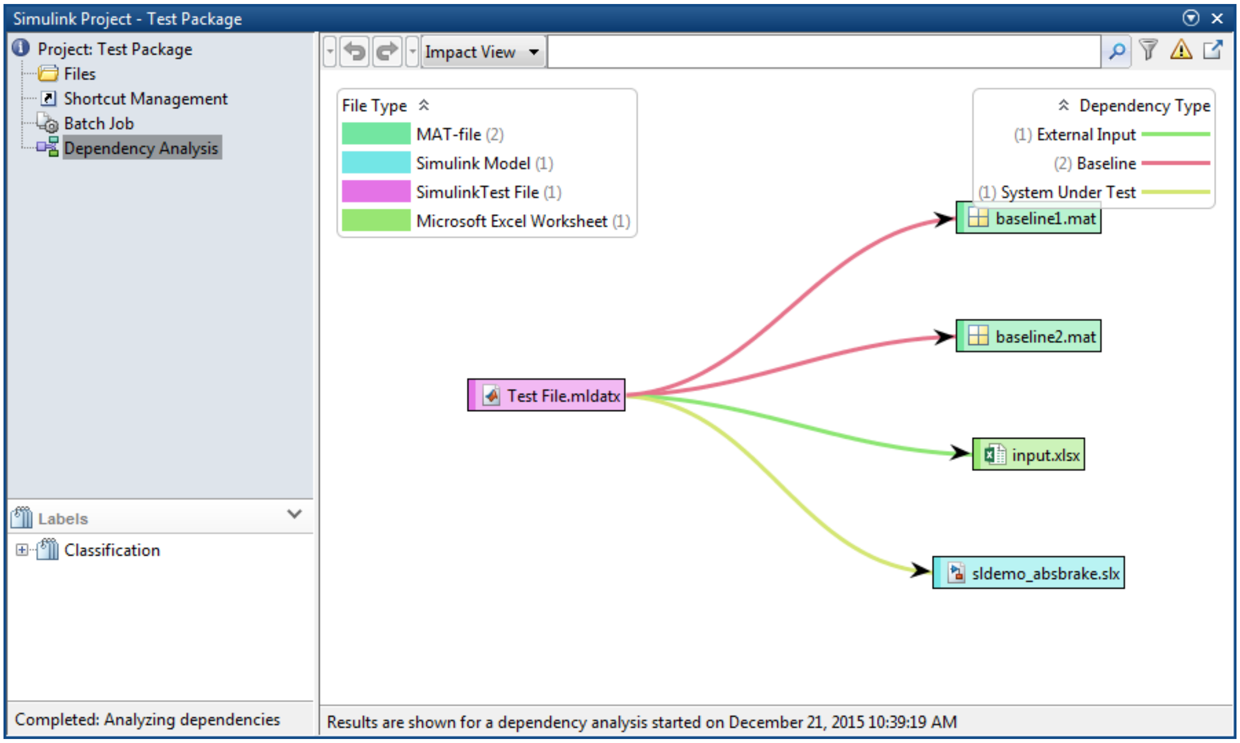
Find Test File Dependencies and Impact

You can find test file dependencies from the Test Browser. Your test file must be saved in a Simulink project.

1 Right-click the test file. Select **Simulink Project > Find Dependencies**.



Dependencies are color coded in the file dependency graph.

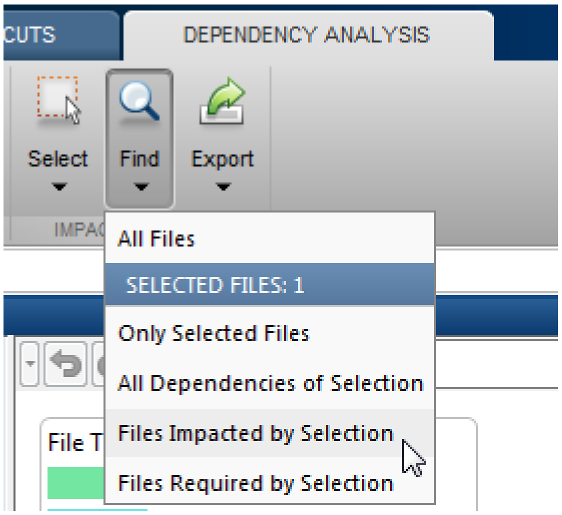


Completed: Analyzing dependencies

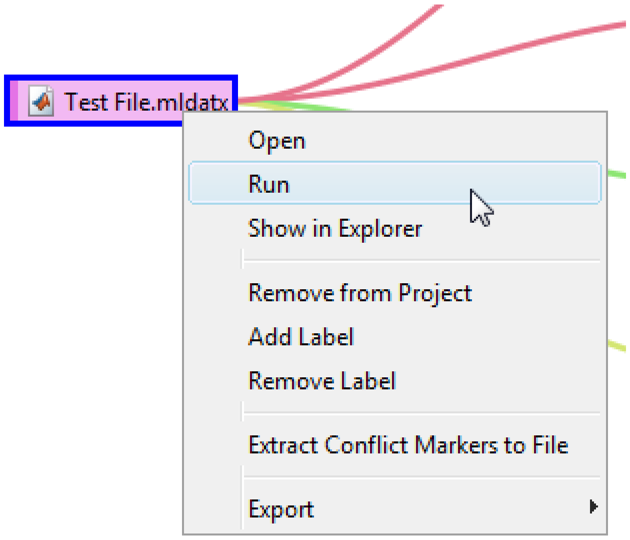
Results are shown for a dependency analysis started on December 21, 2015 10:39:19 AM

If you want to change a model or requirement, you can determine the potential impact of the change on your tests.

1 In the dependency graph, select the item that could impact your tests.
2 In the Simulink Projects toolstrip, click **Files** > **Files Impacted by Selection**.



If you want to run a test file again, then you can right-click the test file in the graph and select **Run**. The Test Manager opens the test file and runs the test cases contained in it.



Share a Test File with Dependencies

You can easily share test files that are already saved in a Simulink Project. If you send the project folder, then it contains the file dependencies for the test file.

See Also

Related Examples

- “What Are Projects?” (Simulink)

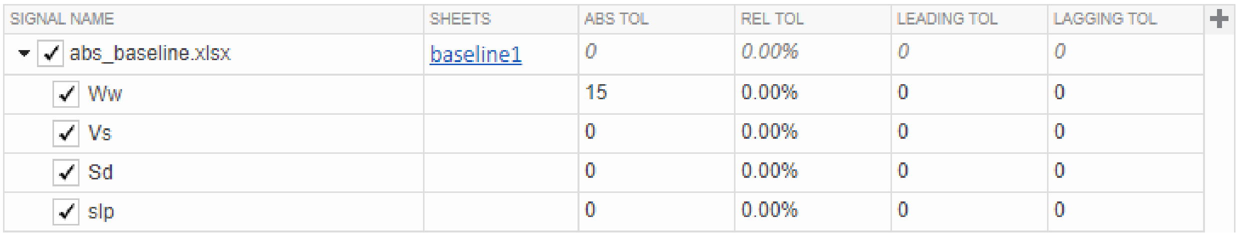
Compare Model Output To Baseline Data

To test the simulation output of a model against a defined baseline, use a baseline test case. In this example, use the `sldemo_absbrake` model to compare the simulation output to a baseline captured from an earlier state of the model.

Create the Test Case

1. Open the `sldemo_absbrake` model.
2. To open the Test Manager from the model, select **Analysis** > **Test Manager**.
3. From the Test Manager toolstrip, click **New** to create a test file. Name and save the test file.

   The test file consists of a test suite that contains one baseline test case. They appear in the **Test Browser** pane.
4. Right-click the baseline test case in the **Test Browser** pane, and select **Rename**. Rename the test case to `Slip Baseline Test`.
5. Under **System Under Test** in the test case, click the **Use current model** button to load the `sldemo_absbrake` model into the test case.
6. To record a baseline from the system under test, under **Baseline Criteria**, click **Capture**.
7. In the Capture Baseline dialog box, for the file format, select `Excel`. Specify a location to save the baseline to and click **Capture**.
8. The baseline criteria file and the logged signals appear in the table. Set the **Absolute Tolerance** of the `Ww` signal to `15`.



SIGNAL NAME	SHEETS	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
✓ abs_baseline.xlsx	baseline1	0	0.00%	0	0
✓ Ww		15	0.00%	0	0
✓ Vs		0	0.00%	0	0
✓ Sd		0	0.00%	0	0
✓ slp		0	0.00%	0	0

Tip To add or remove columns in the baseline criteria table, click the column selector button.

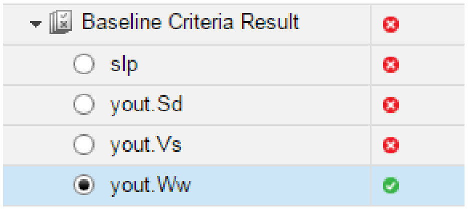
For more information about tolerances and criteria, see "Set Signal Tolerances" on page 7-90.

Run the Test Case and View Results

1. In the `sldemo_absbrake` model, set the **Desired relative slip** constant block to `0.22`.
2. In the Test Manager, select the Slip Baseline Test case in the **Test Browser** pane.
3. On the Test Manager toolstrip, click **Run**.

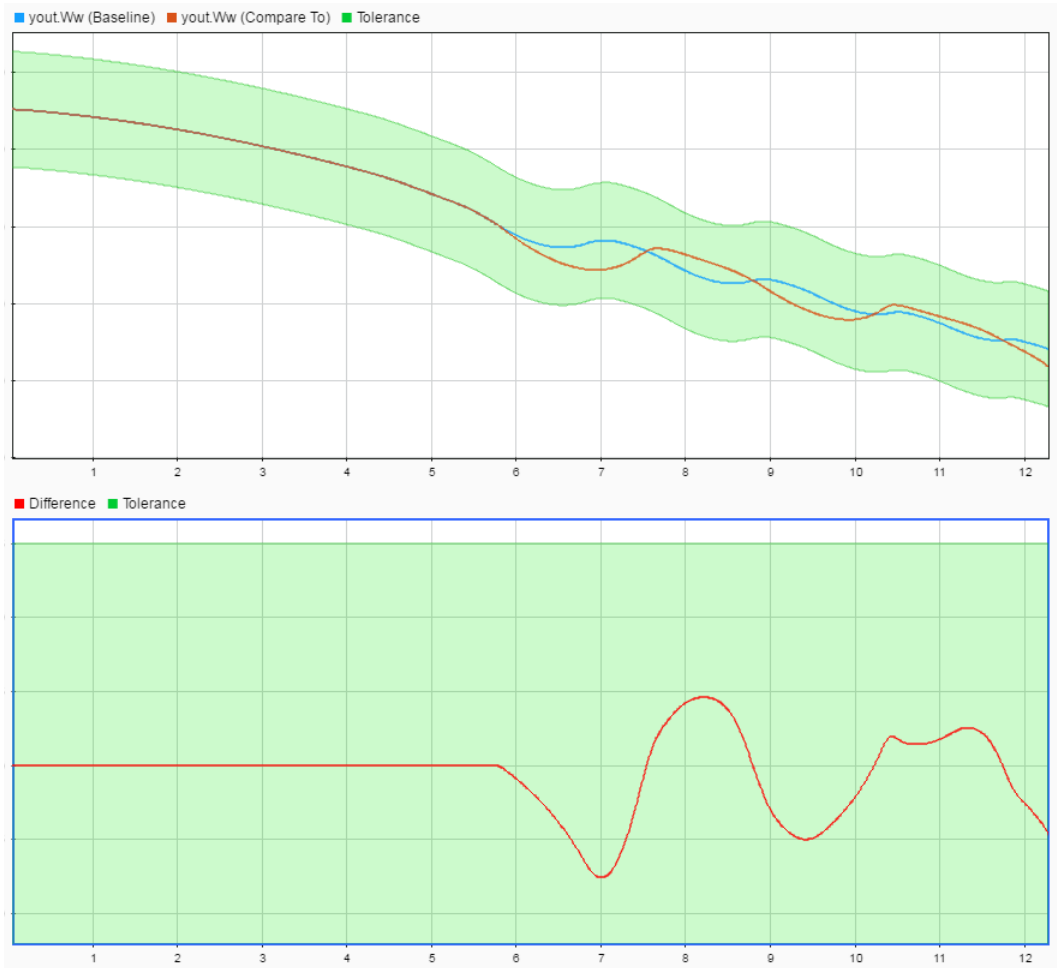
   In the **Results and Artifacts** pane, the new test result appears at the top of the table.
4. Expand the results until you see the baseline criteria result. Right-click the result and select **Expand All Under**.

   The signal `yout.Ww` passes, but the overall baseline test fails because other signal comparisons specified in the **Baseline Criteria** section of the test case were not satisfied.
5. To view the `yout.Ww` signal comparison between the model and the baseline criteria, expand `Baseline Criteria Result` and click the option button next to the `yout.Ww` signal.

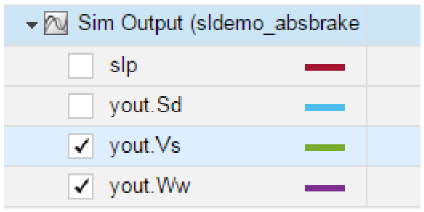
   

Baseline Criteria Result	✗
○ slp	✗
○ yout.Sd	✗
○ yout.Vs	✗
● yout.Ww	✓

   The **Comparison** tab opens and shows the criteria comparisons for the `yout.Ww` signal and the tolerance.



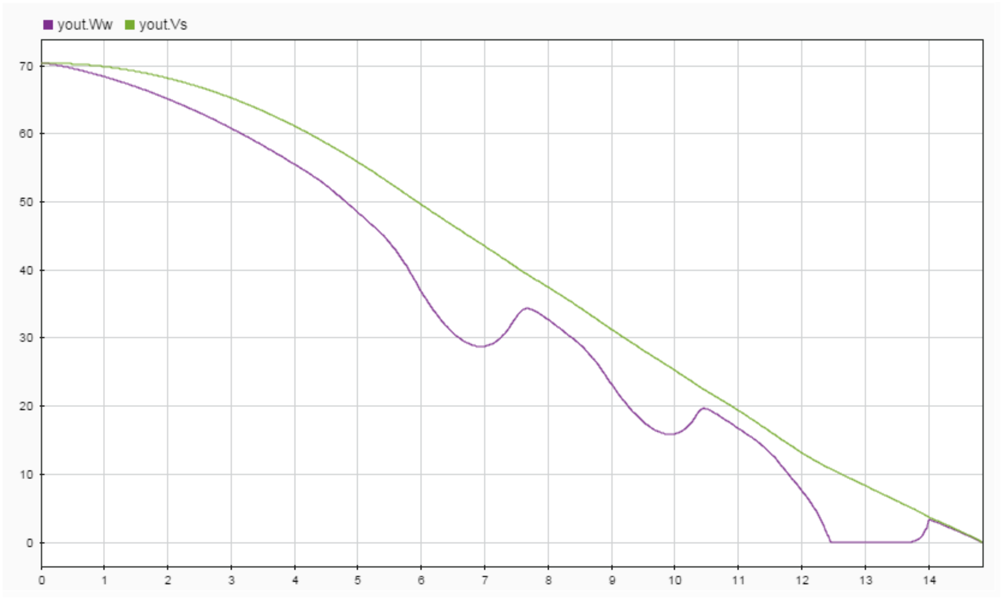
6 You can also view signal data from the simulation. Expand `Sim Output` and select the signals you want to plot.



Sim Output (sldemo_absbrake

☐	slp
☐	yout.Sd
✓	yout.Vs
✓	yout.Ww

The **Visualize** tab opens and plots the simulation output.



For information on how to export results and generate reports from results, see “Export Test Results and Generate Reports” on page 8-9.

See Also

Related Examples

- “Set Signal Tolerances” on page 7-90
- “Capture Baseline Criteria” on page 7-102
- “Run Tests in Multiple Releases” on page 7-55

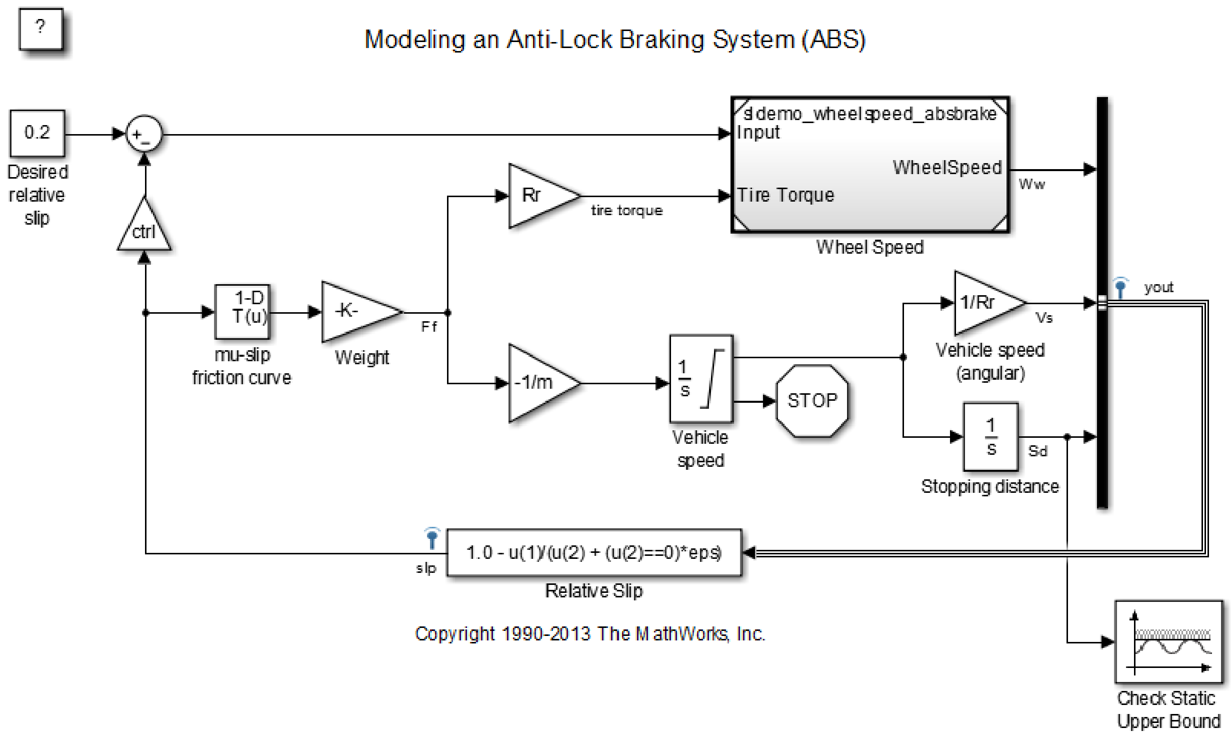
Test a Simulation for Run-Time Errors

In this example, use a simulation test case with the `sldemo_absbrake` model to test for simulation run-time errors.

Configure the Model

Configure the model to check if the stopping distance exceeds an upper bound.

1. Open the model `sldemo_absbrake`.
2. Add the Check Static Upper Bound block from the Model Verification library to the model.
3. Connect the Check Static Upper Bound block to the `Sd` signal.

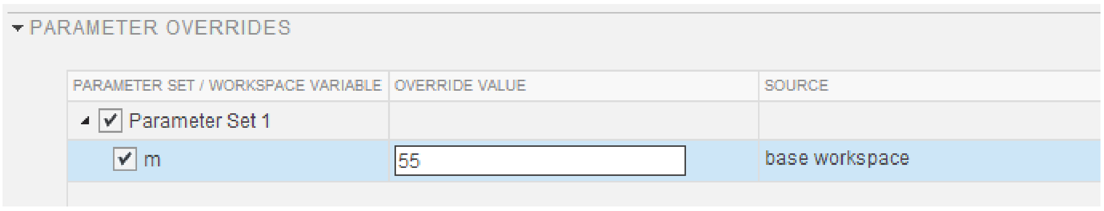


4. In the Check Static Upper Bound block dialog box, and set **Upper bound** to `725`.

Create the Test Case

1. To open the Test Manager, from the model, select **Analysis** > **Test Manager**.
2. To create a test file, click **New**. Name and save the test file.

   The new test file consists of a test suite that contains one baseline test case. They appear in the **Test Browser** pane.
3. Select **New** > **Simulation Test**.
4. Right-click the new simulation test case in the **Test Browser** pane, and select **Rename**. Rename the test case to `Upper Bound Test`.
5. In the test case, under **System Under Test**, click the **Use current model** button to assign the `sldemo_absbrake` model to the test case.
6. Under **Parameter Overrides**, click **Add** to add a parameter set.
7. In the dialog box, click the **Refresh** button to update the model parameter list.
8. Select the check box next to the workspace variable `m`. Click **OK**.
9. Double-click the **Override Value** and enter `55`.



PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE
✓ Parameter Set 1		
✓ m	55	base workspace

   This value overrides the parameter value in the model when the simulation runs.

Note To restore the default value of a parameter, clear the value in the **Override Value** column and press **Enter**.

Run the Test Case

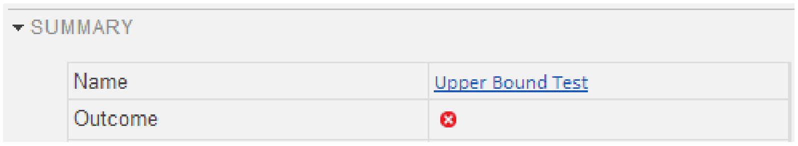
1. In the **Test Browser** pane, select the `Upper Bound Test` case.
2. In the Test Manager toolstrip, click **Run**. The test results appear in the **Results and Artifacts** pane.

View Test Results

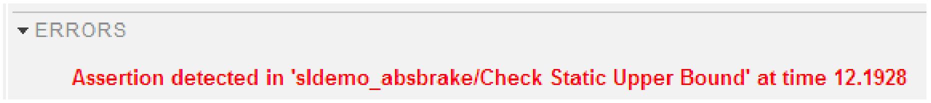
1 Expand the test results, and double-click `Upper Bound Test`.

  A new tab displays the outcome and results summary of the simulation test.

2 The result indicates a test failure. In this case, the stopping distance exceeded the upper bound of 725 and triggered an assertion from the Check Static Upper Bound block. The **Errors** section contains the assertion details.



Name	Upper Bound Test
Outcome	✗



Assertion detected in 'sldemo_absbrake/Check Static Upper Bound' at time 12.1928

See Also

More About

- "Run Tests in Multiple Releases" on page 7-55

Automatically Create a Set of Test Cases

In this section...

Creating Test Cases from Model Elements

You can automatically create a set of test cases that correspond to blocks and test harnesses in your model. You specify whether the test cases are baseline, equivalence, or simulation test cases. To automatically create test cases, your model must contain either or both of the following:

- One Signal Editor or Signal Builder block at the model's top level. One test case is created for each scenario or signal group in the block.
- Test harnesses. If a test harness contains one (and only one) Signal Editor or Signal Builder block at the top level, a test case is created for each scenario or signal group in the block.

To automatically create test cases for your model:

1. In the Test Manager, select **New** > **Test File** > **Test File from Model**.
2. In the dialog box, select the model that you want to generate test cases from. The model must be on the MATLAB path.
3. Select the test case type, and click **Create**.

Generating Test Cases from a Model

Generate test cases based on model hierarchy.

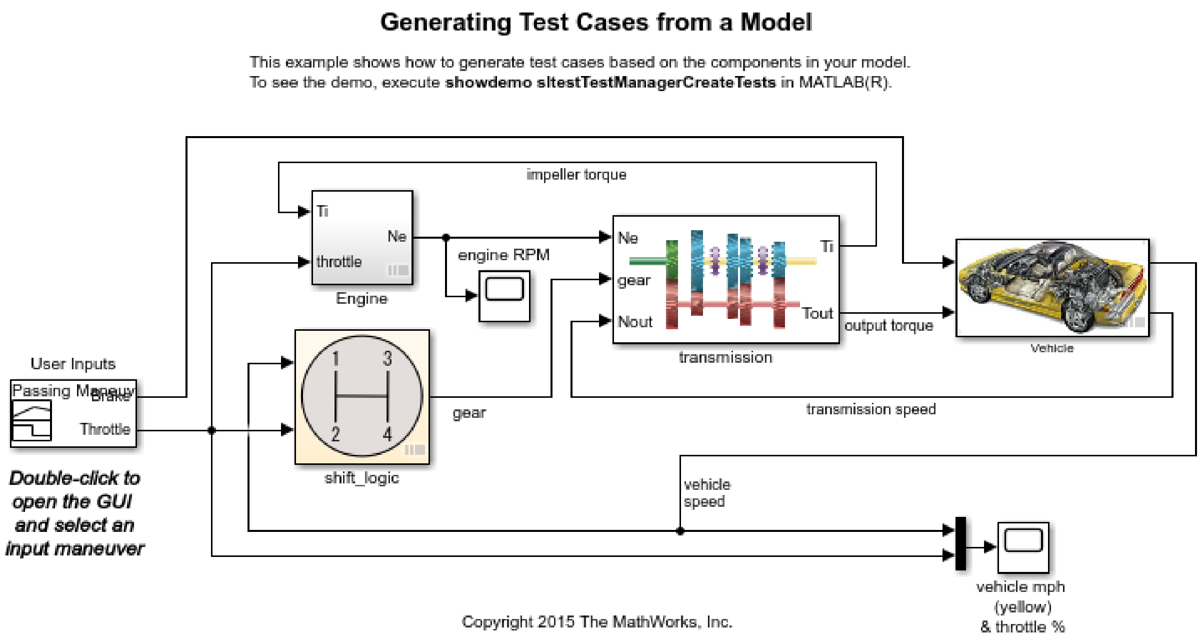
This example shows how to generate test cases based on the components in your model. This example uses the model `sltestTestManagerCreateTestsExample`, which has been pre-configured with the following:

- Signal Builder group in the top model
- Test harnesses in the top model
- Signal Builder group at the top level of a test harness

Open the Model and Test Manager

Execute the following code to open the model configured with different components such as Signal Builder groups and test harnesses.

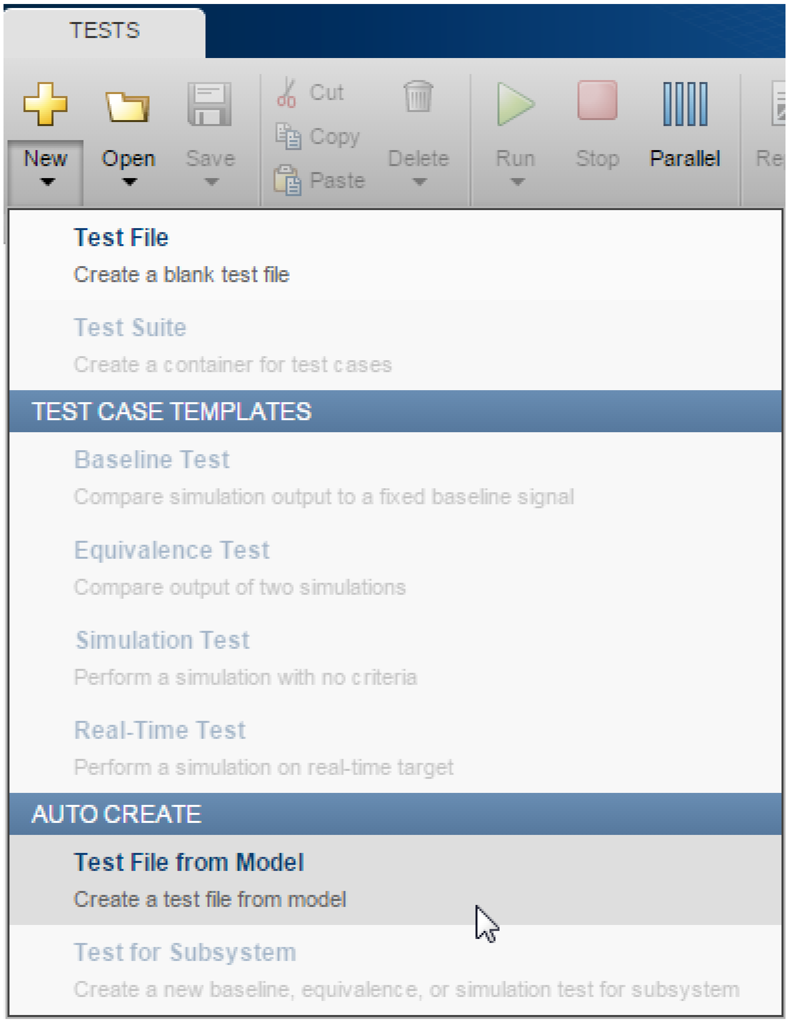
```
mdl = 'sltestTestManagerCreateTestsExample';
open_system(mdl);
```


Generating Test Cases from a Model
This example shows how to generate test cases based on the components in your model.
To see the demo, execute showdemo sltestTestManagerCreateTests in MATLAB(R).
impeller torque
Ti
Ne
throttle
Engine
engine RPM
Ne
gear
Nout
Ti
Tout
transmission
output torque
Vehicle
User Inputs
Passing Maneuv
Brake
Throttle
Double-click to open the GUI and select an input maneuver
1
3
2
4
shift_logic
gear
transmission speed
vehicle speed
vehicle mph (yellow) & throttle %
Copyright 2015 The MathWorks, Inc.

Open the Test Manager. From the Simulink® menu, click **Analysis** > **Test Manager**.

Generate Test Cases From the Model

In the Test Manager, click the **New** arrow and select **Test File > Test File from Model**.



Test File
Create a blank test file

Test Suite
Create a container for test cases

TEST CASE TEMPLATES

Baseline Test
Compare simulation output to a fixed baseline signal

Equivalence Test
Compare output of two simulations

Simulation Test
Perform a simulation with no criteria

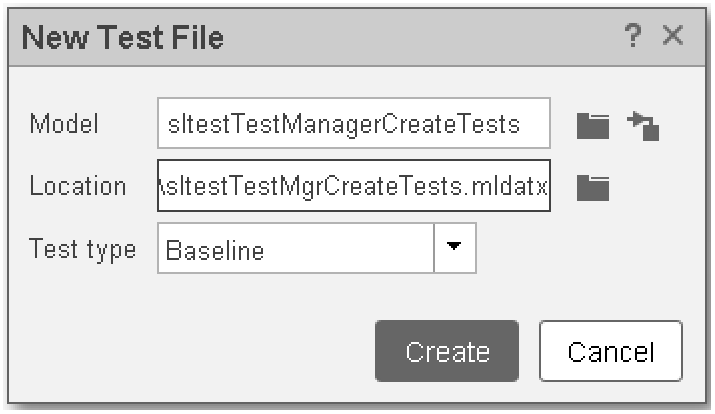
Real-Time Test
Perform a simulation on real-time target

AUTO CREATE

Test File from Model
Create a test file from model

Test for Subsystem
Create a new baseline, equivalence, or simulation test for subsystem

1. In the **New Test File** dialog box, click the **Use current model** button. to specify `sltestTestManagerCreateTestsExample` as the **Model**.
2. Specify the **Location** of the test file.
3. Select the `Baseline` from the **Test Type** dropdown. All test cases generated will be of the test type specified here.
4. Click **Create**.



Model: sltestTestManagerCreateTests

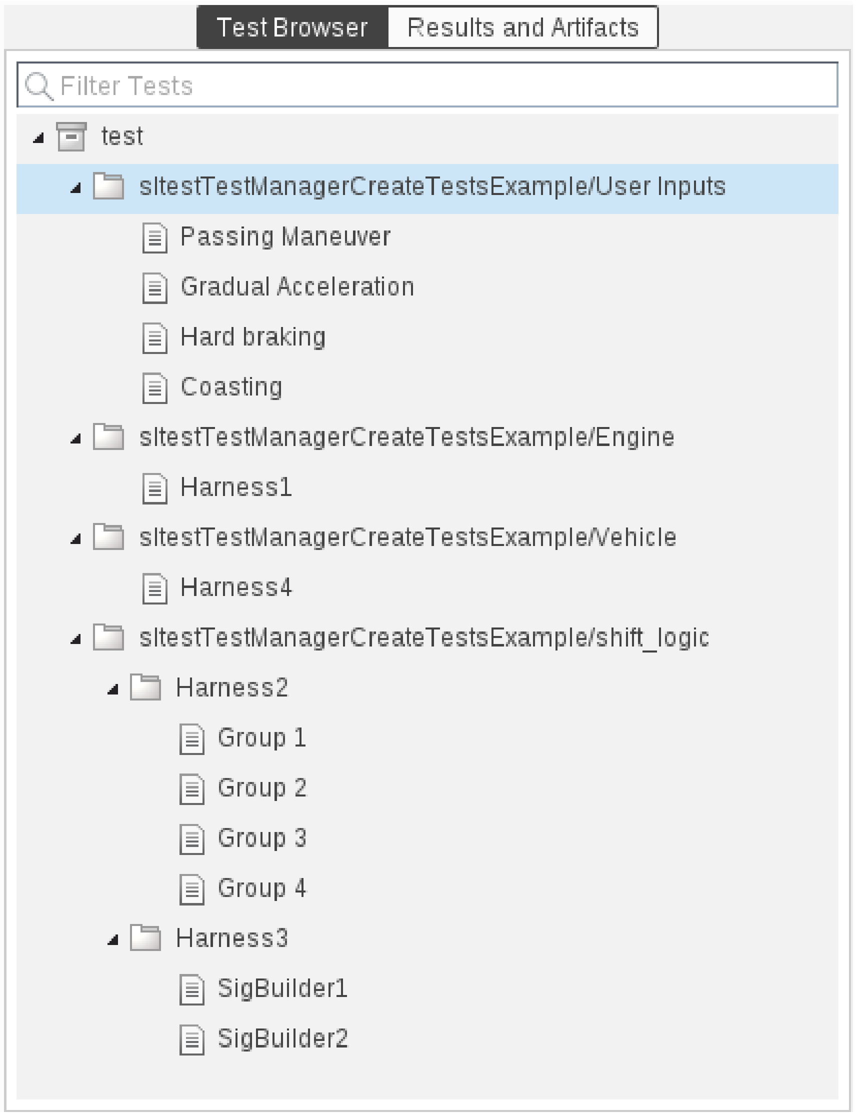
Location: \sltestTestMgrCreateTests.mldatx

Test type: Baseline

The test manager creates a test case for each of the following:

- Signal Builder groups in the top model
- Test harnesses in the top model
- Signal Builder group at the top level of a test harnesses

In each generated test case, you need to specify the comparison criteria, equivalence or baseline, before you run the test.



test
- sltestTestManagerCreateTestsExample/User Inputs
  - Passing Maneuver
  - Gradual Acceleration
  - Hard braking
  - Coasting
- sltestTestManagerCreateTestsExample/Engine
  - Harness1
- sltestTestManagerCreateTestsExample/Vehicle
  - Harness4
- sltestTestManagerCreateTestsExample/shift_logic
  - Harness2
    - Group 1
    - Group 2
    - Group 3
    - Group 4
  - Harness3
    - SigBuilder1
    - SigBuilder2

```
close_system(mdl, 0);
clear mdl;
```

See Also

More About

- “Synchronize Tests” on page 7-24
- “Test Sections” on page 7-95
- “Compare Model Output To Baseline Data” on page 7-9
- “Test a Simulation for Run-Time Errors” on page 7-13
- “Test Two Simulations for Equivalence”
- “Generate Tests for a Subsystem” on page 7-22

Generate Tests for a Subsystem

Use the Test Manager to generate a test case for a subsystem. You can create a baseline, equivalence, or simulation test case. Generating the test case:

- Creates a test harness for the subsystem. The test harness provides a separate simulation environment from the main model.
- Simulates the main model and captures the subsystem input data from the main model simulation.
- For baseline tests, captures output data in a MAT-file or Microsoft Excel file. The output is the test case baseline data.
- Adds the input and output files to the test case.

After you create the test case, configure the test case if you need additional options such as coverage or reports.

You add a generated test case for a subsystem to a test file. If you need a test file to add the test case to, create one.

- Select the test file before you generate the test to add a test suite containing the test case.
- Select a test suite before you generate the test case to add the test case to the test suite. Or, select a test case in a test suite to add the test case to the test suite.

Generate the Subsystem Test Case

You can save the input data to a MAT-file or Excel file. When creating a baseline test, selecting Excel saves the input and output data in the same file. For more information on using Excel files in Test Manager, see “Format Test Case Data in Excel” on page 7-38.

1. In the model that contains the subsystem that you want to create the test for, select the subsystem.
2. In Test Manager, select the test file or test suite that you want to create the test case in, and then select **New > Test for Subsystem**.
3. In the dialog box, click the **Use currently selected subsystem** button to fill in the **Subsystem** and **Top model** fields.
4. Select the test type—baseline, equivalence, or simulation—and specify the file format.

5. Depending on the test type and the file format, you can specify the location for your inputs and your baseline outputs and the sheet name for Excel data.
6. Click **Create**. Test Manager adds a test harness to the subsystem and simulates the model.

   After simulation completes, the test case includes inputs and, for baselines, outputs.

   - For equivalence tests, inputs are added to the test case in the **Inputs** section under **Simulation 1**.
   - For simulation tests, inputs are added under **Inputs**.
   - For baseline tests, inputs are added under **Inputs** and outputs are added under **Baseline Criteria**.
7. Finish configuring the harness and test case for your test scenario.
8. Save the model and the test case.

See Also

More About

- “Test Sections” on page 7-95
- “Compare Model Output To Baseline Data” on page 7-9
- “Test a Simulation for Run-Time Errors” on page 7-13
- “Test Two Simulations for Equivalence”
- “Automatically Create a Set of Test Cases” on page 7-16

Synchronize Tests

If you change the system under test, you can synchronize the test cases to reflect the model changes. Also, if you remove model components, you can disable or delete test cases in the Test Manager when you synchronize.

Synchronizing your test file automatically creates a new test case for:

- Each new scenario in the Signal Editor block at the top level of your model and the top level of each test harness. The model must have only one Signal Editor block at those levels to create a test case.
- Each new signal group in the Signal Builder block at the top level of your model and the top level of each harness. Your model must have only one Signal Block at those levels to create a test case.
- Each new test harness in the model.

To synchronize your test file:

1. In the Test Manager **Test Browser** pane, hover over the test file name that you want to update.
2. Click the synchronization button ⇄ next to the test file name.
3. Follow the prompts to specify:

   - The type of test file to create for the new components
   - Whether to disable or delete out-of-date components

     Disabled tests appear in the list in italic.

See Also

More About

- "Automatically Create a Set of Test Cases" on page 7-16

Run Tests Using External Data

In this section...

You can run test cases using data defined in external MAT-files or Microsoft Excel files. You can map the data to your model (system under test [SUT]) using these mapping modes:

- The names of the inport block the signal data corresponds to
- The full block path name, that is, in the form `system/block`
- The name of the signal associated with the inport block
- Port number, that is, sequential port numbers of the inport blocks, starting at 1

You can add multiple external input files to a test case. After you add the files, select the one you want to use in the test case from the **External Inputs** table. If you are using test iterations, you can assign one input file to each iteration.

For more information about how Simulink handles inport mapping, see “Map Root Inport Signal Data” (Simulink).

Mapping Status

When you map external inputs to model elements, the mapping can create these possible results. These results appear under **Inputs** in the Test Manager interface in the **Status** column:

- Mapped — The mapping succeeded and no further action is needed.
- Failed — The mapping failed. Click the **Failed** link for more information.
- Warning — The mapping occurred with warnings. Click the **Warning** link to see whether you need to address them

- Stale — This status can occur when you update your external inputs in Test Manager. A stale state occurs if you did not map the new inputs. To address this status, click the **Status** link, which opens the Add Input dialog box. Click **Map Inputs** to map the new input data and then click **Add**.

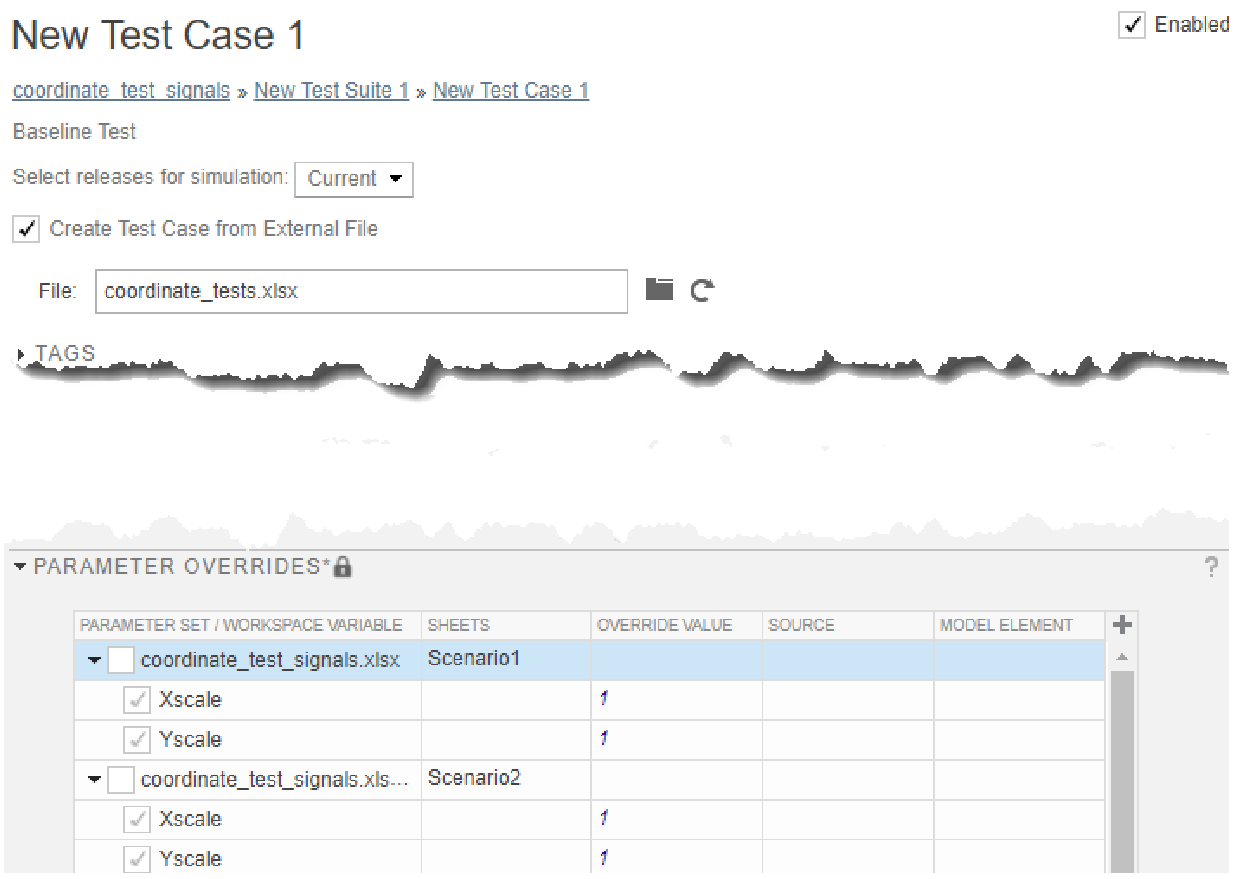
Create a Test Case from an Excel Spreadsheet

You can create a test case in Test Manager using the Create Test from Spreadsheet wizard. From Simulink Test Manager, select **New > Test from Spreadsheet**. Select **Use existing test data from a spreadsheet** and follow the prompts. You can use the following spreadsheet and model as an example:

```
<matlabroot>\examples\simulinktest\coordinate_tests.xlsx
coordinate_transform_test.slx
```

In the **Attributes** page, make sure all attribute categories that exist in the spreadsheet are displayed. Click **Validate** to map each input to the model by block name. If necessary, make changes to the spreadsheet and/or SUT and click **Refresh** and validate again. After a successful validation, save the test.

- The test case imports the spreadsheet. The fields defined in the spreadsheet are locked to the spreadsheet, and cannot be edited in the Test Manager.


New Test Case 1

✓ Enabled

coordinate_test_signals » New Test Suite 1 » New Test Case 1

Baseline Test

Select releases for simulation: Current

✓ Create Test Case from External File

File: coordinate_tests.xlsx

TAGS

PARAMETER OVERRIDES*

PARAMETER SET / WORKSPACE VARIABLE	SHEETS	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
coordinate_test_signals.xlsx	Scenario1			
✓ Xscale		1		
✓ Yscale		1		
coordinate_test_signals.xls...	Scenario2			
✓ Xscale		1		
✓ Yscale		1		

To change the locked fields, edit the spreadsheet outside of MATLAB.

Import an Excel Spreadsheet into a Test Case

If you have a test case and want to add test data to it from Excel spreadsheet, you must associate it with the spreadsheet:

1 Open the test case.
2 Check the **Create Test Case from External File** option.
3 Browse for the spreadsheet with the test data.

The input, parameter, and comparison signal data in the spreadsheet overrides the data in the test case. The fields defined in the spreadsheet are locked to the spreadsheet. To edit, do one of the following:

- Edit the spreadsheet outside of MATLAB and click **Refresh** for the **File** field.
- Clear the **Create Test Case from External File** option and edit the test case in the Test Manager. Note that selecting this option again causes values in the spreadsheet to overwrite the values in the test.

Add Microsoft Excel File as Input

You can import Microsoft Excel spreadsheets to use as inputs. You can import multiple sheets at once and specify a range of data. Selecting sheets and specifying ranges is useful when each sheet contains a different data set or the same file contains input data and expected outputs.

For information about the Excel file format, see "Format Test Case Data in Excel" on page 7-38.

1. In the test case, expand the **Inputs** section and click **Add**.
2. Browse to your Microsoft Excel file and click **Add**.
3. Select each sheet that contains input data. You can specify a range of data.
4. If you want to use each sheet to create an input set in the table, select **Create scenarios from each sheet**.
5. Under **Input Mapping**, select a mapping mode.
6. Click **Map Inputs**. The **Mapping Status** table shows the port and signal mapping.

   For more information about troubleshooting the mapping, see "Understand Mapping Results" (Simulink).
7. Click **Add**.

Importing Microsoft® Excel® Data

Test a model using inputs stored in Microsoft Excel.

This example shows how to create a test case in the Test Manager and map data to the test case from a Microsoft® Excel® file. Input mapping supports Microsoft Excel spreadsheets only for Microsoft Windows®.

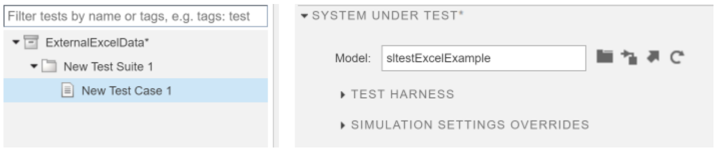
Create a Test File

1. Open the Test Manager. Enter

```
sltest.testmanager.view
```

2. In the Test Manager toolbar, select **New > Test File**. Save the file to a writable directory. The Test Manager creates a test file with an empty baseline test case.

3. In the test browser, select the test case. In the test editor, under the **System Under Test** section, enter `sltestExcelExample`.



Model: sltestExcelExample

Configure the External Inputs.

1. Expand the **Inputs** section of the test case.

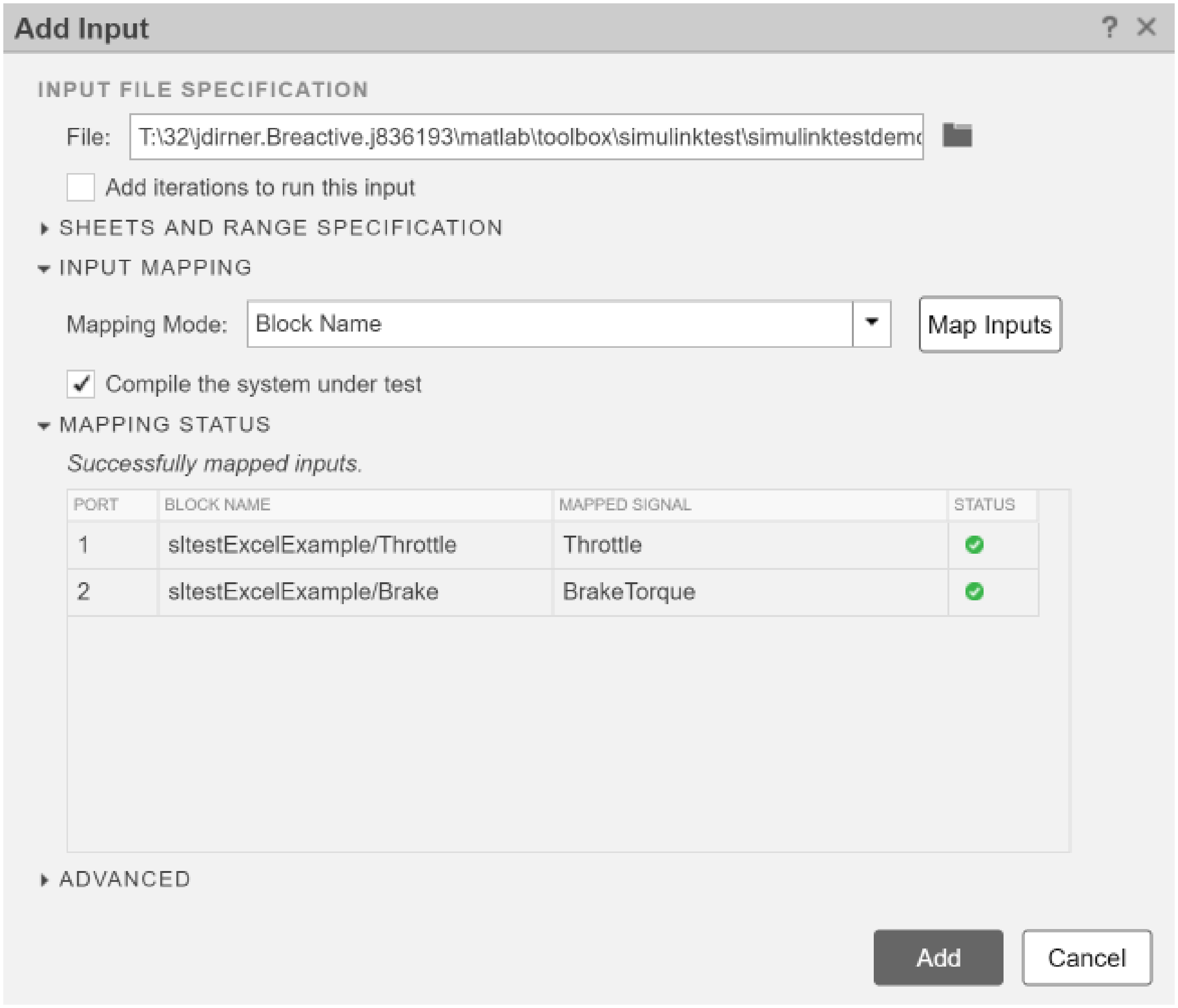
2. To include the input data in the test results, click **Include external inputs/signal builder data in test result**.

3. Under the **External Inputs** table, click **Add**.

4. In the **Add Input** dialog box, for **File**, select the `sltestExampleInputs.xlsx` from the `matlab/toolbox/simulinktest/simulinktestdemos` directory.

5. In the **Add Input** dialog box,

- Select the **Acceleration** sheet from the sheets table.
- Select **Mapping Mode** : `Block Name`.
- Click **Map Inputs**.
- Click **Add**.



INPUT FILE SPECIFICATION

File: T:\32\jdirner.Breactive.j836193\matlab\toolbox\simulinktest\simulinktestdem

☐ Add iterations to run this input

SHEETS AND RANGE SPECIFICATION

INPUT MAPPING

Mapping Mode: Block Name

☑ Compile the system under test

MAPPING STATUS

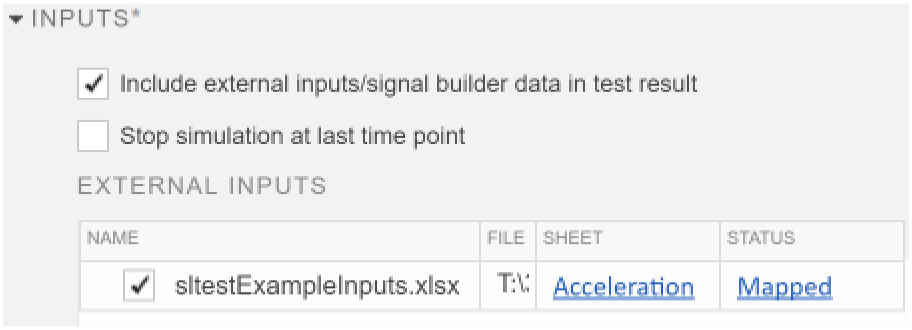
Successfully mapped inputs.

PORT	BLOCK NAME	MAPPED SIGNAL	STATUS
1	sltestExcelExample/Throttle	Throttle	✓
2	sltestExcelExample/Brake	BrakeTorque	✓

ADVANCED

The **Mapping Mode** controls the method used to map data from the Microsoft Excel sheet to root-level Inport blocks in the model. For more information, see Use External Inputs in Test Cases.

The test case shows the inputs mapped.



INPUTS*

[✓] Include external inputs/signal builder data in test result

[ ] Stop simulation at last time point

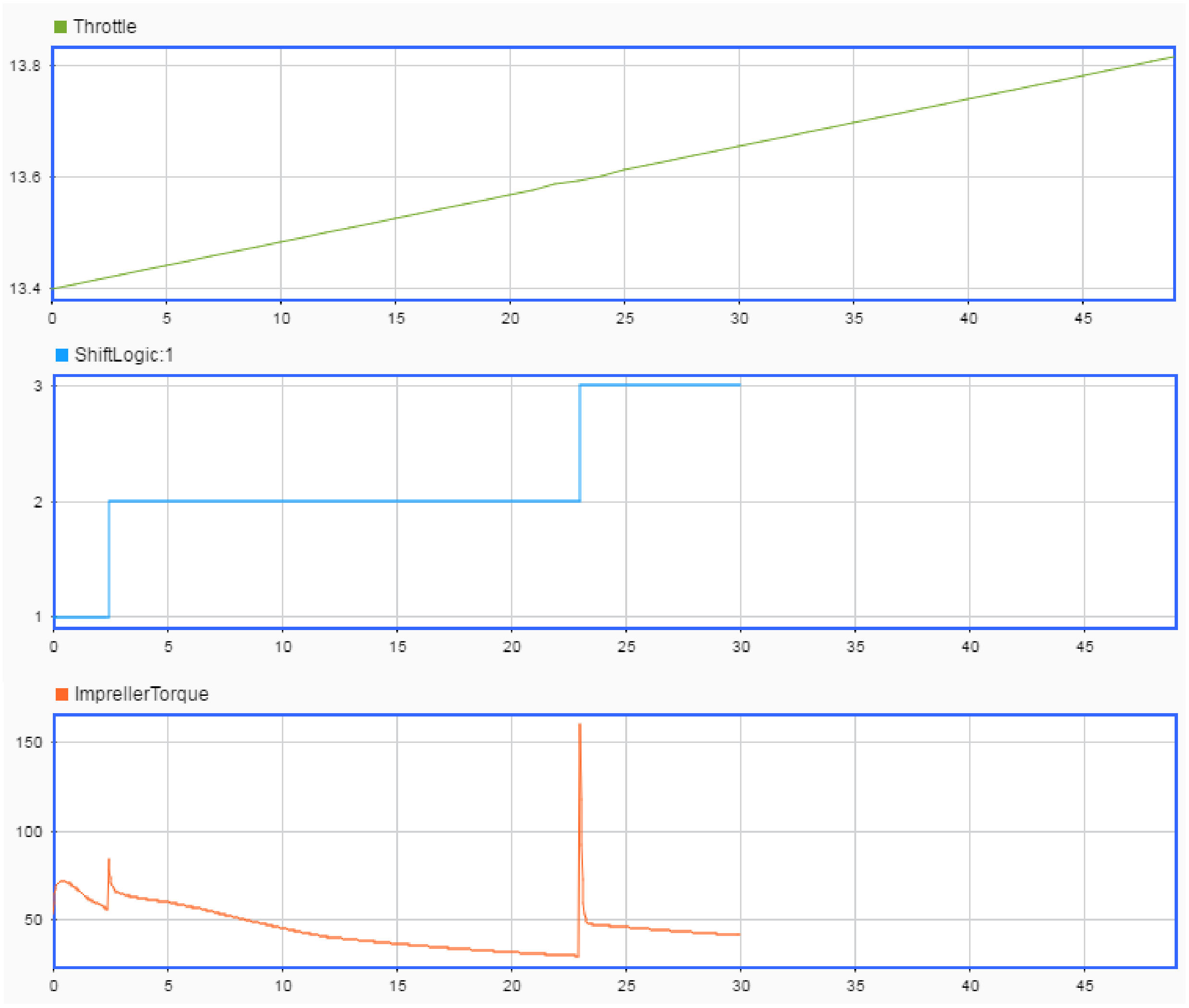
EXTERNAL INPUTS

NAME	FILE	SHEET	STATUS
✓ sltestExampleInputs.xlsx	T:\	Acceleration	Mapped

Run the Test

1. In the toolbar, click **Run**.

2. In the **Results and Artifacts** pane, you can plot signals from the external inputs or the simulation output.



Add a MAT-File as an External Input

1 In the test case, expand the **Inputs** section and click **Add**.
2 Browse to the MAT-file and click **Add**.
3 Under **Input Mapping**, choose a mapping mode.

4 Click **Map Inputs**. The **Mapping Status** table shows the port and signal mapping.

   For information about troubleshooting the mapping status, see “Understand Mapping Results” (Simulink).

5 Click **Add**.

See Also

`sltest.testmanager.TestInput`

More About

- “Map Signal Data to Root Inports” (Simulink)
- “Map Root Inport Signal Data” (Simulink)
- “Test Case Input Data Files” on page 7-34

Test Case Input Data Files

You can use Test Manager to create MAT-file and Microsoft Excel data files to use as inputs to test cases. You generate a template that contains the signal names and the times, and then enter the data.

Creating a data file also adds the file to the list of available input files for the test case. After you add input data, you can then select the file to use in your test case.

You can create files for input data only for tests that run in the current release. To select the release, in the test case, use the **Select releases for simulation** list.

You can edit input files. After you create the template, select the file from the list of input files and click **Edit**. MAT-files open in the signal editor. Excel files open in Excel.

Selecting the **Add an iteration that runs this input** check box adds an iteration to the test case under **Table Iterations** and assigns the input file to it. After you create the input file, continue to specify the iteration. For more information on iterations, see “Test Iterations” on page 7-71.

Generate an Excel Template

You can generate a template test spreadsheet from a model or harness (system under test [SUT]). You can then complete the spreadsheet with external data and import it into Simulink Test as a test case.

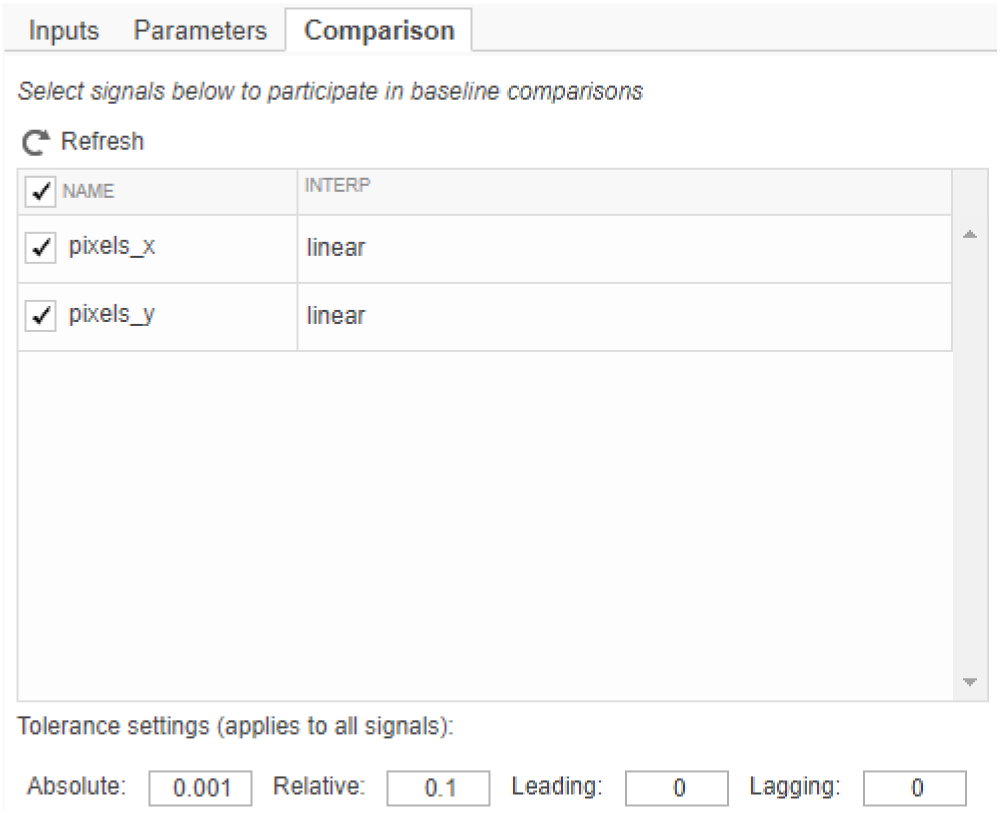
The Create Test from Spreadsheet wizard parses the SUT for test attributes and automatically generates a template spreadsheet and a test case:

- Inputs — Inputs are characterized by root input ports
- Parameters — Named parameters in the model
- Comparison signals — Logged signals and output ports

The wizard allows you to filter and edit the attributes needed for testing. The resulting spreadsheet has separate column sets for inputs, parameters, and comparison signals. If multiple iterations are required, a separate sheet in the same file is generated for each scenario. You can expand the spreadsheet to add time-based signal data, tolerances, and parameter overrides. See “Format Test Case Data in Excel” on page 7-38 for the full description of the format readable by Simulink Test.

You can use the model `coordinate_transform_test` as an example for the process. The model must be on the MATLAB path.

1. Open the test manager using **Analysis > Test Manager**.
2. Open the wizard. From Simulink Test Manager, select **New > Test from Spreadsheet**. Select **Create a test template file for specifying data** and follow the prompts.
3. In the **Attributes** page, select which attribute categories are to be included in the spreadsheet. For example, if parameter overrides are not necessary for the tests, clear Parameters. The attribute categories shown on the page are derived from the SUT. Comparison signals are always shown.
4. If the test requires all attributes in a category as is, select **Yes, include all attributes in the spreadsheet** and click **Next**. If not, select **No, I want to filter and edit the attributes**. This shows a page with a tab for each attribute category.
5. If you are filtering the attributes, in the **Parameters** and **Comparison** tabs, clear the attributes that are not needed. For example, you can remove a logged signal from this list if it is not to be used for comparison in the tests.
6. Optionally change tolerances in the **Comparison** page. The tolerance settings apply to all signals in the list. To specify different tolerances for each signal, edit the spreadsheet after it is generated.



Select signals below to participate in baseline comparisons

✓ NAME	INTERP
✓ pixels_x	linear
✓ pixels_y	linear

Tolerance settings (applies to all signals):

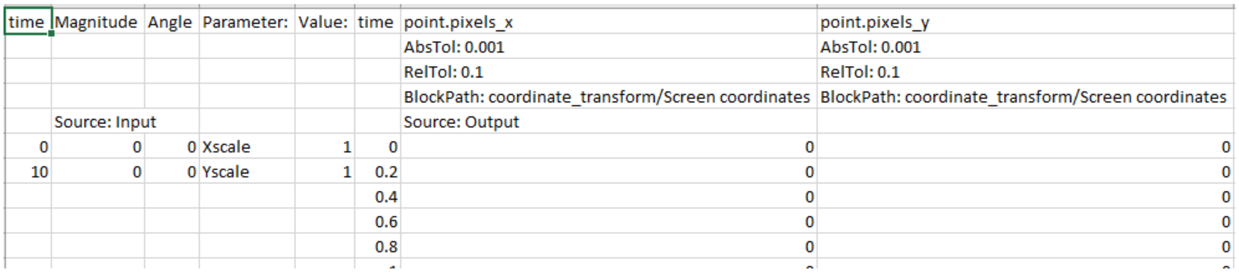
Absolute: 0.001 Relative: 0.1 Leading: 0 Lagging: 0

If you change the SUT during the selection process, click **Refresh** to synchronize the attribute lists with the SUT. Once selection is complete, click **Next** and keep following the prompts.

7 In the **Scenarios** page, specify the number of test scenarios and a base name for the sheets in the spreadsheet.

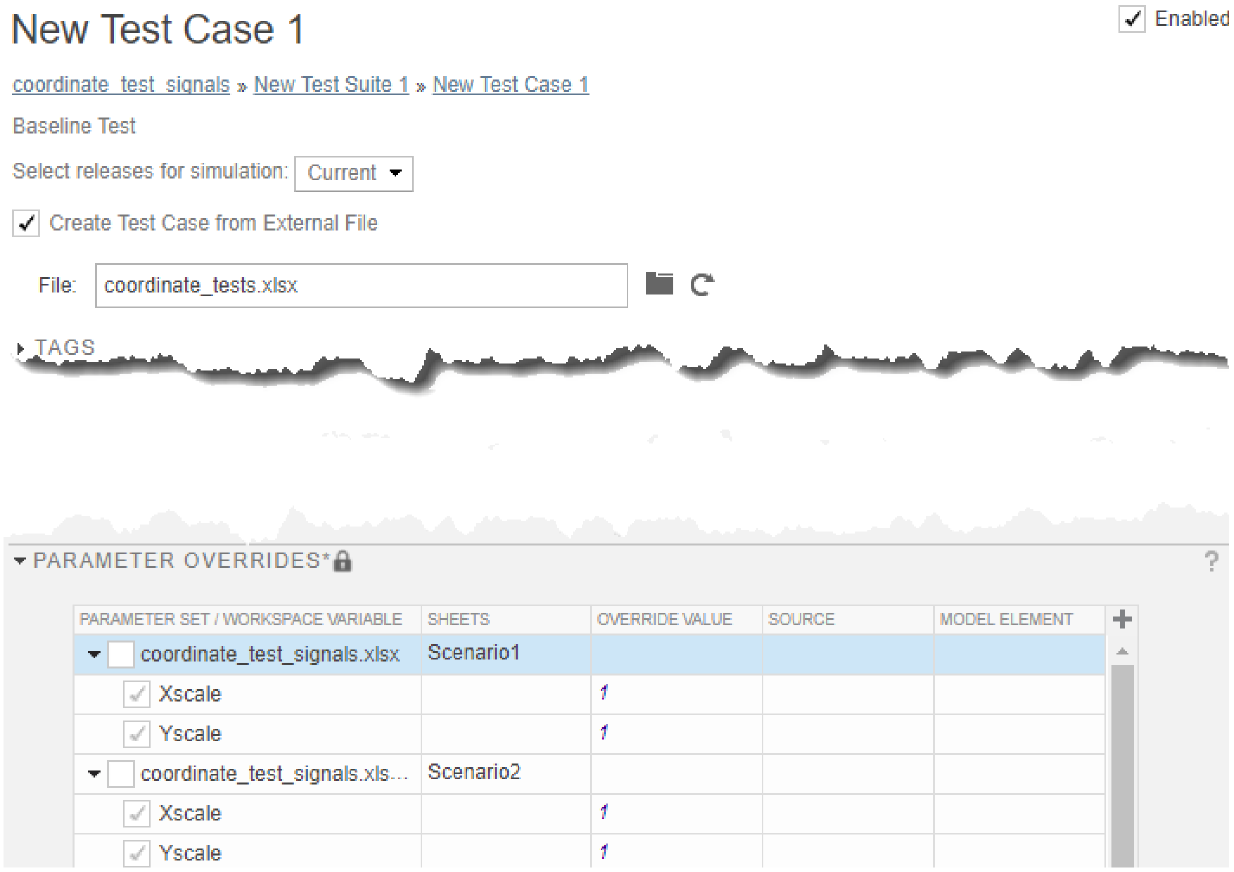
If comparison signals are selected, the wizard runs the model to capture the baseline. Make sure that the model does not run indefinitely by setting a finite stop time. The wizard creates two files:

- Excel spreadsheet — The spreadsheet includes columns for inputs, parameters, and comparison signals. Inputs and comparisons have different time bases. An identical sheet for each test scenario is generated. Complete the spreadsheet outside MATLAB to uniquely define each scenario.



time	Magnitude	Angle	Parameter:	Value:	time	point.pixels_x	point.pixels_y
						AbsTol: 0.001	AbsTol: 0.001
						RelTol: 0.1	RelTol: 0.1
						BlockPath: coordinate_transform/Screen coordinates	BlockPath: coordinate_transform/Screen coordinates
	Source: Input					Source: Output	
0	0	0	Xscale	1	0	0	0
10	0	0	Yscale	1	0.2	0	0
					0.4	0	0
					0.6	0	0
					0.8	0	0

- Test file — The test case imports the Excel spreadsheet. The fields defined in the spreadsheet are locked to the spreadsheet, and cannot be edited in the Test Manager.



New Test Case 1

✓ Enabled

coordinate_test_signals » New Test Suite 1 » New Test Case 1

Baseline Test

Select releases for simulation: Current

✓ Create Test Case from External File

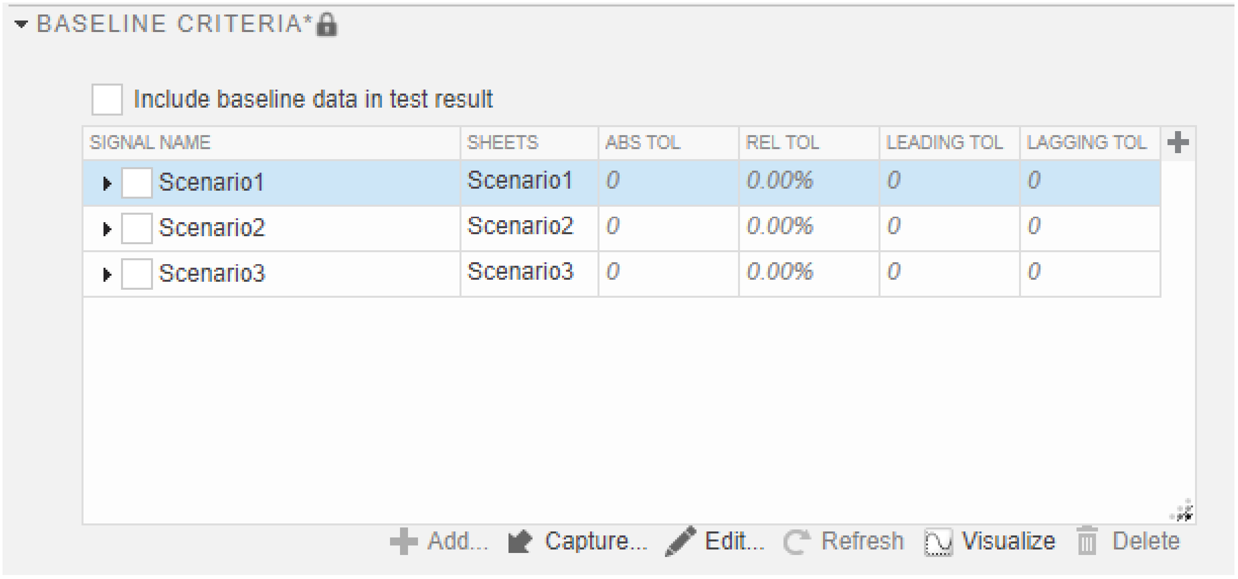
File: coordinate_tests.xlsx

TAGS

PARAMETER OVERRIDES*

PARAMETER SET / WORKSPACE VARIABLE	SHEETS	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
coordinate_test_signals.xlsx	Scenario1			
✓ Xscale		1		
✓ Yscale		1		
coordinate_test_signals.xls...	Scenario2			
✓ Xscale		1		
✓ Yscale		1		

To change the locked fields, edit the spreadsheet outside MATLAB.If you change a parameter, you must capture the baseline again by clicking the **Capture** button.



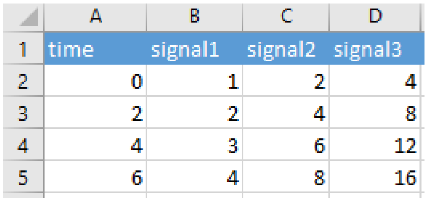
SIGNAL NAME	SHEETS	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
Scenario1	Scenario1	0	0.00%	0	0
Scenario2	Scenario2	0	0.00%	0	0
Scenario3	Scenario3	0	0.00%	0	0

Format Test Case Data in Excel

You can specify signal data in a Microsoft Excel file to use as input to your test case or as baseline criteria (outputs). The Excel file includes time and signal data. To support a range of models and configurations, you can specify signal data of most data types. For exceptions, see “Limitations” on page 7-47. You can indicate whether signals are scalar, multidimensional, or complex. You can optionally specify the data type, block path and port index, units, interpolation type, and function-call execution times.

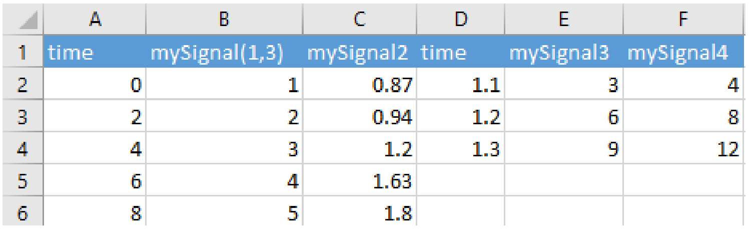
Basic Excel File Format

The figure shows the basic format of the Excel file. This example uses scalar signals and the default data type, `double`, for all signals.



	A	B	C	D
1	time	signal1	signal2	signal3
2	0	1	2	4
3	2	2	4	8
4	4	3	6	12
5	6	4	8	16

- When specifying time and signal data (and not function-call execution times), the `time` column is the first column. Time values must increase in value, and every cell must contain a value. .
- Include one column for each input signal in the first row. In each column, include the signal data for each time point. Signal names are case-sensitive.
- If dataset elements have different time vectors, the spreadsheet can have more than one time column. In this case, the columns to the right of each time column, up to the next time column, define signals along that time vector. The signal columns must have the same number of rows as the time column they define values for. The figure shows an example that has a time column for each time vector.

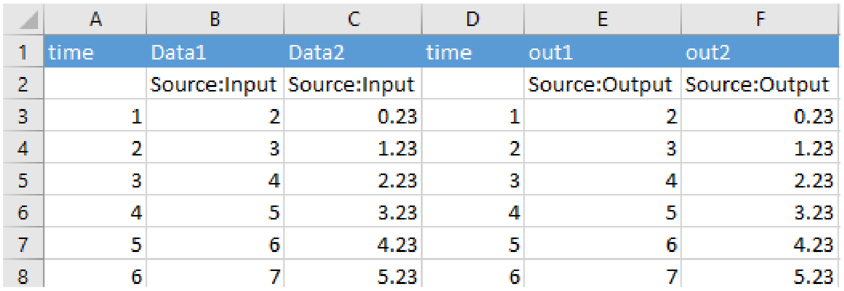


	A	B	C	D	E	F
1	time	mySignal(1,3)	mySignal2	time	mySignal3	mySignal4
2	0	1	0.87	1.1	3	4
3	2	2	0.94	1.2	6	8
4	4	3	1.2	1.3	9	12
5	6	4	1.63			
6	8	5	1.8			

When you import data, you specify the mapping mode. To map using signal or block names, add the block or signal names and qualifiers in the first row. If you are mapping using block path and name, also specify them in the optional rows (see “Block Path and Port Index” on page 7-43). If you are mapping using port numbers, signal columns map to the model ports in order during import, ignoring the block or signal names.

Input and Output Data

You can save inputs and outputs in the same Excel file in the same sheet. Specify whether the signals are for inputs or outputs in one of the optional rows, using `Source:Input` or `Source:Output` as the label. Keep all the inputs together and all the outputs together.



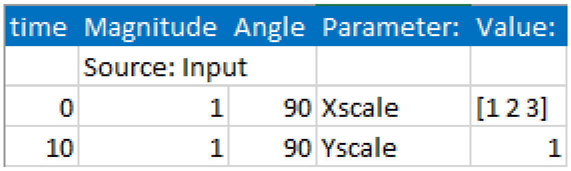
	A	B	C	D	E	F
1	time	Data1	Data2	time	out1	out2
2		Source:Input	Source:Input		Source:Output	Source:Output
3	1	2	0.23	1	2	0.23
4	2	3	1.23	2	3	1.23
5	3	4	2.23	3	4	2.23
6	4	5	3.23	4	5	3.23
7	5	6	4.23	5	6	4.23
8	6	7	5.23	6	7	5.23

To import the file as input data, use the **Inputs** section of the test case, described in “Run Tests Using External Data” on page 7-25. To use the Excel file as expected outputs, select it to add as baseline data, in the **Baseline Criteria** section of the test case, described in “Baseline Criteria” on page 7-102.

When you capture inputs and expected outputs in Test Manager, you can save inputs and outputs to the same Excel file. Both sets of data are saved to the same sheet unless you specify a different sheet. Saving the inputs or expected outputs adds the file to the test. See “Capture Baseline Criteria” on page 7-102.

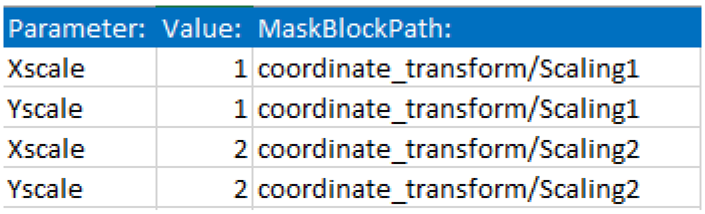
Parameters

You can save parameter override values in the same Excel file in the same sheet. Specify the name of the parameter in the `Parameter:` column and the override value in the `Value:` column. Each row corresponds to a parameter. Enter vector and matrix parameters as you would in MATLAB. You can also use MATLAB expressions for parameter values. Values are read as strings and are evaluated at runtime. Signal data and parameter data start on the same row.



time	Magnitude	Angle	Parameter:	Value:
	Source: Input			
0	1	90	Xscale	[1 2 3]
10	1	90	Yscale	1

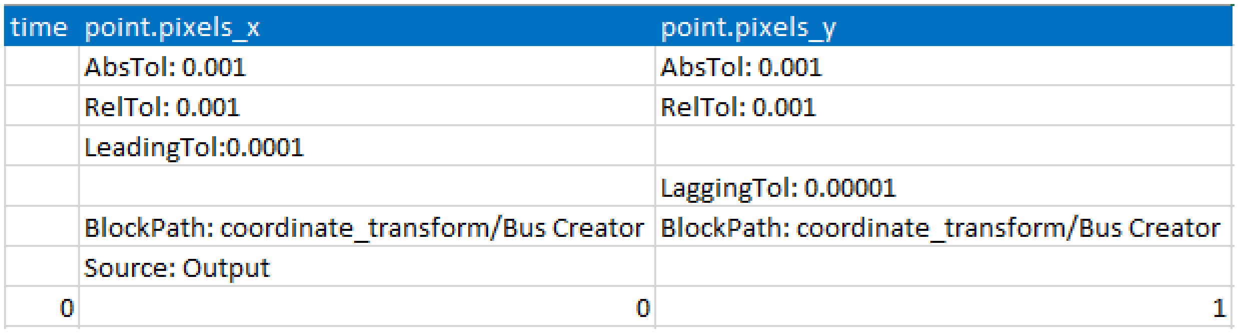
The `Parameter` column must precede the `Value` column and both columns are required. For a parameter in a masked subsystem, add a third column `MaskBlockPath` and enter the path to the block.



Parameter:	Value:	MaskBlockPath:
Xscale	1	coordinate_transform/Scaling1
Yscale	1	coordinate_transform/Scaling1
Xscale	2	coordinate_transform/Scaling2
Yscale	2	coordinate_transform/Scaling2

Signal Tolerances

You can specify tolerances for comparison signals in the Excel file. It is possible to define one or more tolerance types for each signal, in any order. Prefix the tolerance value with one of: `AbsTol:`, `RelTol:`, `LeadingTol:`, `LaggingTol:`. The format is `<ToleranceType>:<ToleranceValue>`.

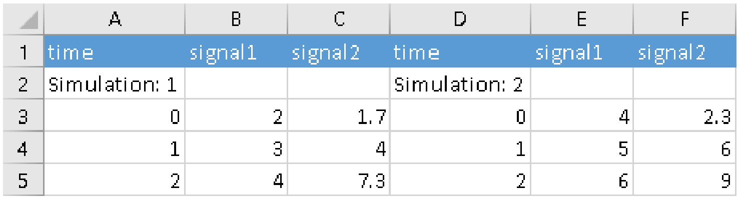


time	point.pixels_x	point.pixels_y
	AbsTol: 0.001	AbsTol: 0.001
	RelTol: 0.001	RelTol: 0.001
	LeadingTol:0.0001	
		LaggingTol: 0.00001
	BlockPath: coordinate_transform/Bus Creator	BlockPath: coordinate_transform/Bus Creator
	Source: Output	
0	0	1

Tolerances are interpreted as floating-point doubles. Each tolerance type should be in a dedicated row. Once a row is declared to be a certain type of tolerance, all columns in that row must be of that type. An empty cell is treated as tolerance of zero. For more information on tolerances, see “Compare Model Output To Baseline Data” on page 7-9.

Simulation for Equivalence Tests

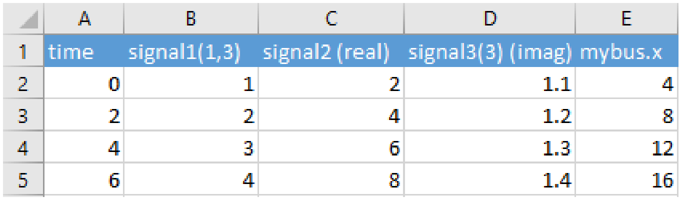
When you perform equivalence tests in Test Manager, you compare the results of two simulations. If your Excel input file is for an equivalence test, you can specify the inputs for each simulation. Specify the simulation in one of the optional rows, using `Simulation:1` or `Simulation: 2` as the label. Keep the inputs for each simulation together.



	A	B	C	D	E	F
1	time	signal1	signal2	time	signal1	signal2
2	Simulation: 1			Simulation: 2		
3	0	2	1.7	0	4	2.3
4	1	3	4	1	5	6
5	2	4	7.3	2	6	9

Scalar, Multidimensional, Complex, and Bus Signals

In addition to scalar signal names, you can indicate multidimensional, complex, and bus signals, or a combination of these. The figure shows some examples.



	A	B	C	D	E
1	time	signal1(1,3)	signal2 (real)	signal3(3) (imag)	mybus.x
2	0	1	2	1.1	4
3	2	2	4	1.2	8
4	4	3	6	1.3	12
5	6	4	8	1.4	16

Specify block paths, and optionally, port index in one of the optional rows. See "Block Path and Port Index" on page 7-43.

Base Name

Names are case-sensitive.

Multidimensional Signals

Use parentheses with the signal dimension after the signal name. For example:

- `mySignal(1,3)`

Dimensions on the signal that you do not specify default to zeros of the same data type and complexity as the dimensions that you specify.

Complex Signals

For complex signals, use `(real)` or `(imag)` with the signal name. For example:

- `mySignal (real)`
- `mySignal (imag)`
- `mySignal(1,3) (imag)`

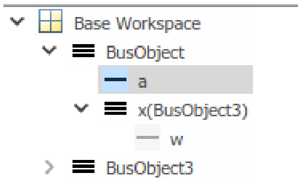
If you do not specify a real counterpart to an imaginary number, the real value defaults to zeros.

Bus Signals

Specify bus signals in the form `signalname.busElement.nestedElement` for as many nested elements as the bus has. For example:

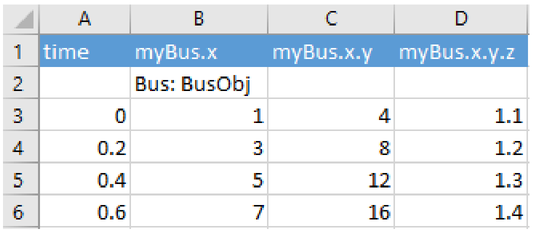
- `myBusSignal.x`
- `busSignal2.x.z`

Suppose the inport block `myBus` is a bus object with this structure:



In this case, specify the signal as `myBus.a.w`.

The figure shows an example of specifying bus signals with a bus data type (see “Data Type” on page 7-44). The `BusObj` data type also applies to the columns to the right because the base name for these signals is the same.

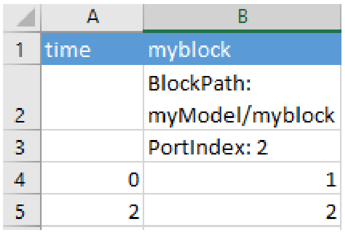


	A	B	C	D
1	time	myBus.x	myBus.x.y	myBus.x.y.z
2		Bus: BusObj		
3	0	1	4	1.1
4	0.2	3	8	1.2
5	0.4	5	12	1.3
6	0.6	7	16	1.4

Block Path and Port Index

If you want to specify the signal using the block path, enter it in one of the optional rows in the form `BlockPath:` *`path to block`*. When you specify a block path, you can also specify the block port index. The default port index is 1. Enter the port index in the row following the block path in the form `PortIndex:` *`port number`*. For example:

- `BlockPath: mymodel/myblock`
- `PortIndex: 2`



	A	B
1	time	myblock
2		BlockPath: myModel/myblock
3		PortIndex: 2
4	0	1
5	2	2

Data Type, Unit, Interpolation, and Block Path/Port Index

In the optional rows between the signal name and the time and signal data, you can include any combination of information about the signal:

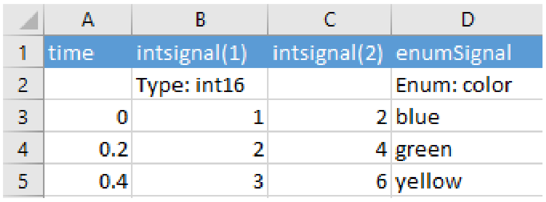
- Data type
- Units
- Interpolation
- Block path and port index (see “Block Path and Port Index” on page 7-43)

Data Type

Enter data types in the row after the signal name. The default data type is `double`.

You can mix data types in the same row, but you must use the same data type for all columns of a multidimensional or complex signal.

You can leave the columns to the right of the data type declaration empty if that data type applies to the signal data in those columns. For example, here the data type `int16` applies to columns B and C because they are dimensions of the same signal.



	A	B	C	D
1	time	intsignal(1)	intsignal(2)	enumSignal
2		Type: int16		Enum: color
3	0	1	2	blue
4	0.2	2	4	green
5	0.4	3	6	yellow

Built-In MATLAB Data Types

Specify built-in MATLAB data types supported in Simulink in the form `Type:` *`data type`*. See “Data Types Supported by Simulink” (Simulink). For example:

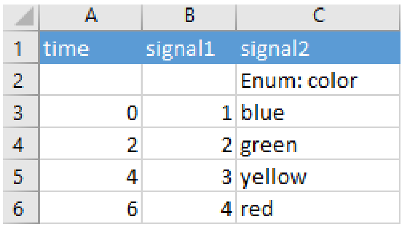
- `Type: int16`
- `Type: uint32`

Enumerations

Specify an enumeration data type in the form `Enum:` *`class`*. For example:

- `Enum: school`

The data in the cells correspond to enumerated values. For example:



	A	B	C
1	time	signal1	signal2
2			Enum: color
3	0	1	blue
4	2	2	green
5	4	3	yellow
6	6	4	red

`Enum` data type dimensions that do not have data default to the default enumeration value.

Fixed Point

Indicate a fixed-point data type using the prefix `Fixdt:`, followed by the data type in one of these forms:

- A `fixdt` constructor, for example, `Fixdt: fixdt(1,16)`.
- A unique data type name string, for example, `Fixdt: sfix16_B7`. To learn about specifying data type names, see “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).
- A `numerictype` object in the base workspace, for example, `Fixdt: mytype`.

Bus

Specify the bus object in the form `Bus:` *`bus object`* with a bus signal. For example:

- `Bus: BusObject1`

To specify a bus signal, see “Bus Signals” on page 7-43.

Alias

Specify an alias data type in the form `Alias:` *`alias type`*. To learn about alias data types, see `Simulink.AliasType`.

Units

Optionally, include a row for units. Specify units in the form `Unit:` *`units`*. You can specify units and physical quantity. For example:

- `Unit: g`
- `Unit: kg@mass`

Interpolation

Optionally, include a row for interpolation. The default is linear. Specify interpolation as `Interp: zoh` or `Interp: linear`.

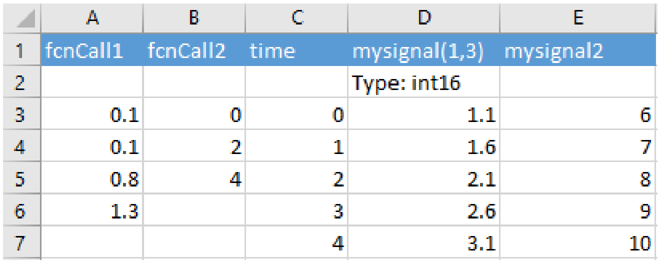
Synchronization

Optionally, include a row for synchronization. The default is `union`. Specify synchronization as `Sync: union` or `Sync: intersection`.

Function-Call Execution Times

If the model contains control signals for function-call subsystems, add columns for each one before the first `time` column. Enter the control signal name in the column heading. Enter the points of time when you want to execute the function call in the column.

Function-call execution times that you specify are independent of the times in the `time` column. The figure shows how to format two function-call blocks that execute at various times. The time and signal data and data type information are independent of the function-call information.



	A	B	C	D	E
1	fcnCall1	fcnCall2	time	mysignal(1,3)	mysignal2
2				Type: int16	
3	0.1	0	0	1.1	6
4	0.1	2	1	1.6	7
5	0.8	4	2	2.1	8
6	1.3		3	2.6	9
7			4	3.1	10

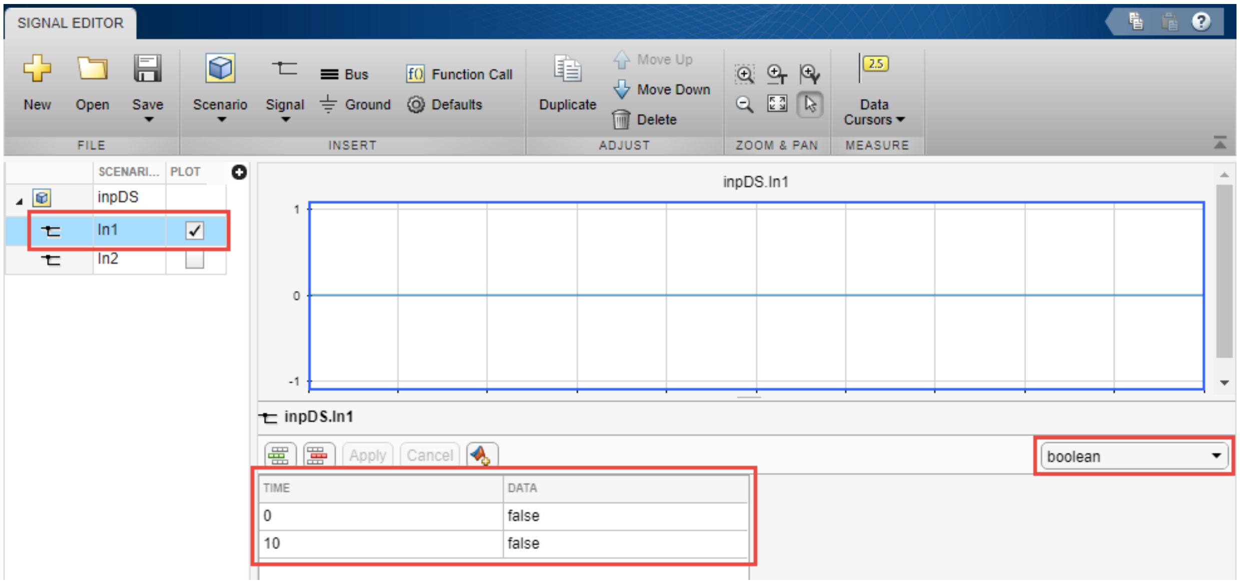
Limitations

Arrays of buses as a data type are not supported.

Create a MAT-File for Input Data

1. In the test case, under **System Under Test**, specify the model whose input data you want to create a MAT-file for.
2. In the **Inputs** section of the test case, click **Create**.
3. In the dialog box, set the file format to `MAT-file`. Specify the location for the MAT-file and click **Create**.

   The signal editor opens.
4. In the **Scenarios and Signals** pane of the signal editor, expand the data node. Then select the signal whose data you want to add.
5. Specify the signal data. Select the data type from the list, and enter the time and signal data for the signal.



TIME	DATA
0	false
10	false

6 To update your signal data, click **Apply**.

7 After adding the signal data, click **Save**.

See Also

`sltest.testmanager.BaselineCriteria` | `sltest.testmanager.TestInput`

More About

- "Run Tests Using External Data" on page 7-25
- "Select Releases for Testing" on page 7-95
- "Run Tests Using External Data" on page 7-25
- "Baseline Criteria" on page 7-102

Capture Simulation Data in a Test Case

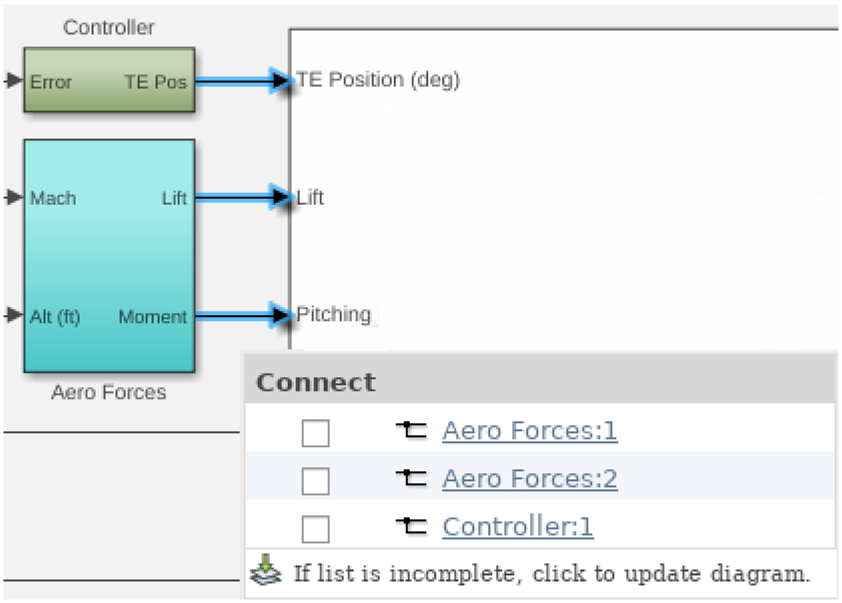
Capture signal data in your test results by adding signals to the **Simulation Outputs** section of the test case. Each output is called a logged signal. Signals listed in **Simulation Outputs** appear in the test results along with signals that are already selected for logging in Simulink.

You can use logged signals for data comparison in baseline criteria, equivalence criteria, custom criteria, and for data visualization in the Simulation Data Inspector. Logged signals enable you to further test your Simulink model without changing the model. In addition to signals from the top model, you can also log signals from subsystems and model references. You can select signals associated with local and global data store memory, and from data store memory that uses a `Simulink.Signal` object.

Add Logged Signals in the Test Manager

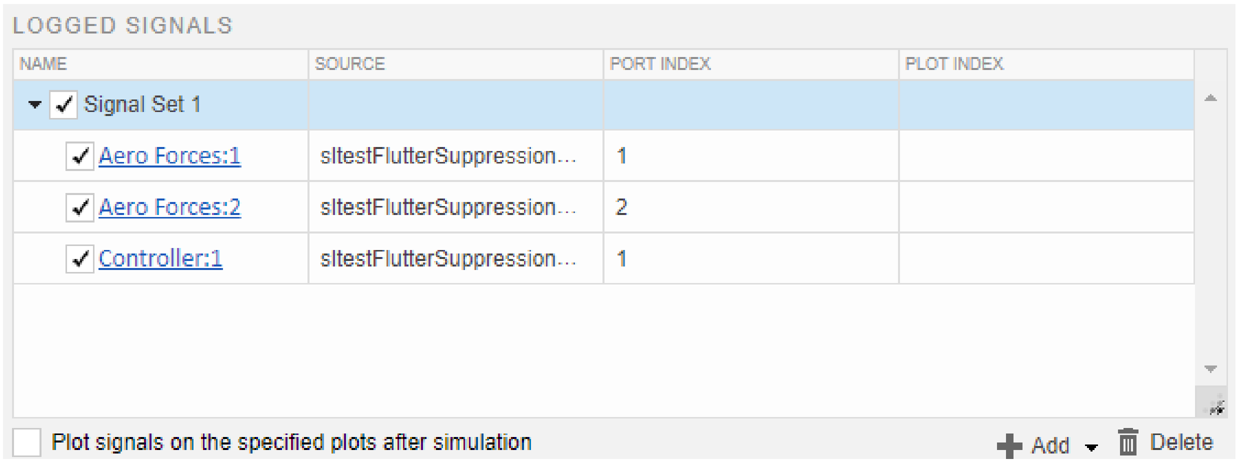
To add signals:

1. Open the model `sltestFlutterSuppressionSystemExample`.
2. Under **Simulation Outputs**, click **Add**.
3. In the system under test, highlight blocks or signals that you want to log. To select multiple items, click and drag a selection box over multiple items.
4. A dialog box appears. Select signals in the dialog box.



5 Continue adding signals to the test case. Each time you select a signal, the dialog box also shows previously logged signals. You can remove a signal from logging by clearing the selection.

6 The signals appear in the **Logged Signals** table in the test case.


LOGGED SIGNALS

NAME	SOURCE	PORT INDEX	PLOT INDEX
✓ Signal Set 1			
✓ Aero Forces:1	sltestFlutterSuppression...	1	
✓ Aero Forces:2	sltestFlutterSuppression...	2	
✓ Controller:1	sltestFlutterSuppression...	1	

7 To add a signal set, click the **Add** arrow and select **Signal Set**.

8. To specify a specific plot for a signal, enter a number in the **Plot Index** column. By default, the signals appear on one plot.

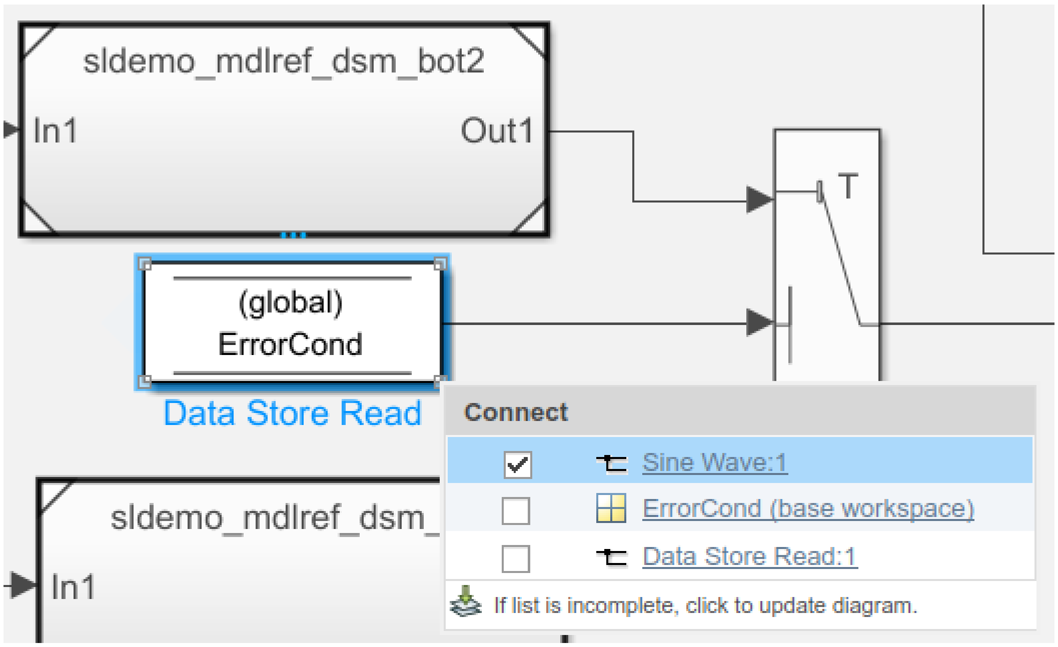
   You can specify to display the plot immediately after running the test by selecting the **Plot signals on the specified plots after simulation** check box.

After you run the test, the logged signals appear in the test case results under **Sim Output**. Select each signal to display on the plot. If you specify a plot index, the signal appears in the plot number you specified.

Capture Data from Local and Global Data Stores

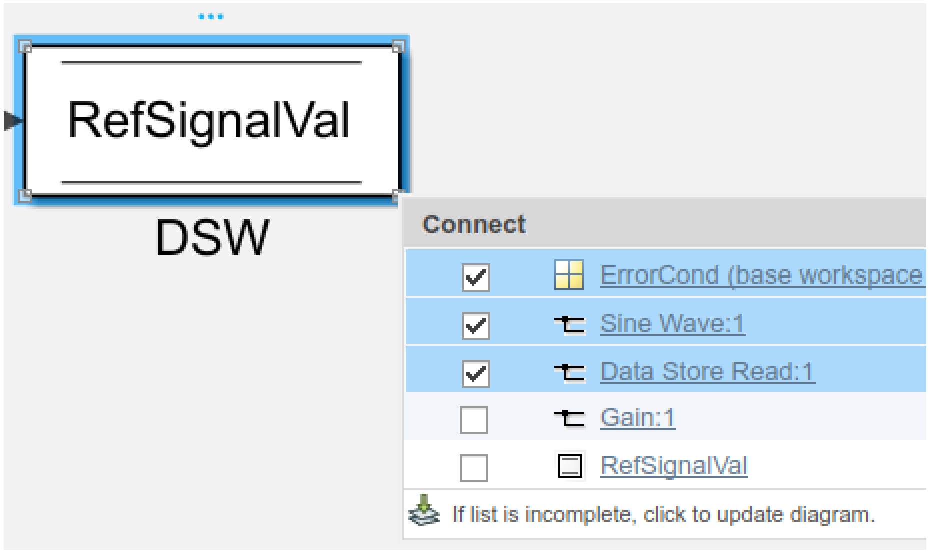
Perform similar steps to add simulation output associated with data store memory:

1. Open the model `sldemo_mdlref_dsm`, which contains local and global data store memory.
2. In a test case, from the top model, add the Sine Wave block for logging.
3. Click on the Data Store Read block in the top model. Click on the **click to update diagram** box. The dialog box displays the signal associated with the block and the data associated with the `Simulink.Signal` object in the base workspace. The model displays the signal storage class for the block, `(global)`.



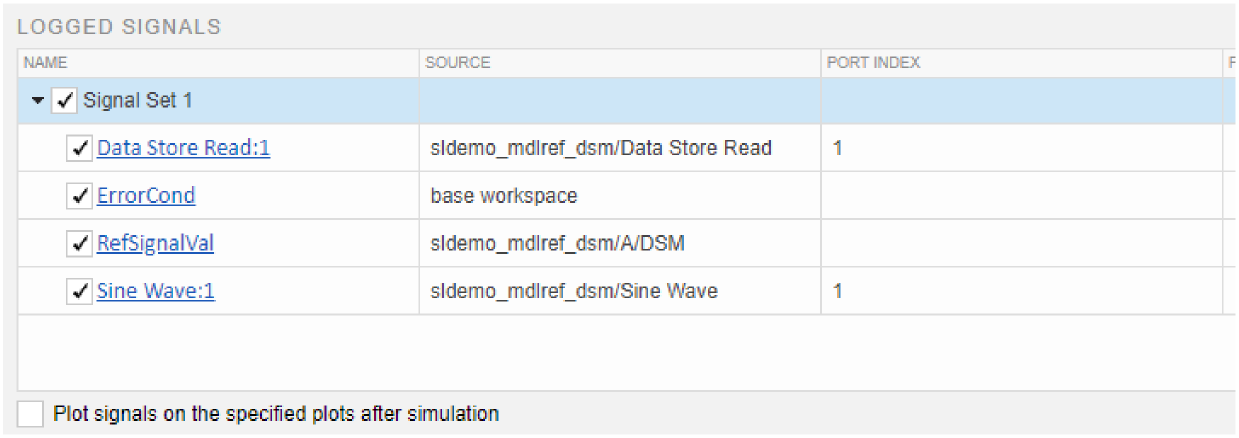
4 Select both signals in the dialog box.

5 Double-click the model reference `sldemo_mdlref_dsm_bot` to open it, and then open the subsystem `PositiveSS`. Select the Data Store Write block. The table displays the input signal from the Gain block and the data store memory, `RefSignalVal`.


RefSignalVal
DSW
Connect
ErrorCond (base workspace
Sine Wave:1
Data Store Read:1
Gain:1
RefSignalVal
If list is incomplete, click to update diagram.

6 Select the `RefSignalVal` data store memory for logging. The dialog box uses a different icon to indicate the data store memory.

7 Finish selecting signals by clicking **Done** in the Test Manager window. In the Test manager, the signals appear under **Logged Signals**. The **Source** column displays the full path information for each signal. For the signal associated with the `Simulink.Signal` object, **Source** displays the workspace location of the `Simulink.Signal` object.



LOGGED SIGNALS

NAME	SOURCE	PORT INDEX
✓ Signal Set 1		
✓ Data Store Read:1	sldemo_mdlref_dsm/Data Store Read	1
✓ ErrorCond	base workspace	
✓ RefSignalVal	sldemo_mdlref_dsm/A/DSM	
✓ Sine Wave:1	sldemo_mdlref_dsm/Sine Wave	1

Plot signals on the specified plots after simulation

See Also

`Simulink.Signal` | `sltest.testmanager.LoggedSignal` | `sltest.testmanager.LoggedSignalSet` | `sltest.testmanager.TestCase`

More About

- “Assess Simulation and Compare Output Data” on page 3-10
- “Compare Model Output To Baseline Data” on page 7-9

Run Tests in Multiple Releases

If you have more than one release of MATLAB installed, you can run tests in multiple releases. This option lets you run tests in releases that do not have Simulink Test, starting with R2011b.

While you can run test cases on models in previous releases, the release you run the test in must support the features of the test. If, for example, your test involves test harnesses or test sequences, the release must support those features for the test to run.

Before you can create tests that use additional releases, add them to your list of available releases using Test Manager preferences. See “Add Releases Using Test Manager Preferences” on page 7-56.

Considerations for Testing in Multiple Releases

Testing Models in Previous or Later Releases

Your model or test harness must be compatible with the MATLAB version running your test.

- If you have a model created in a newer version of MATLAB, to test the model in a previous version of MATLAB, export the model to a previous version and simulate the exported model with the previous MATLAB version. For more information, see the information on exporting a model in “Save a Model” (Simulink).
- To test a model in a more recent version of MATLAB, consider using the Upgrade Advisor to upgrade your model for the more recent release. For more information, see “Consult the Upgrade Advisor” (Simulink).

Test Case Compatibility with Previous Releases

When performing testing in multiple releases, the MATLAB version must support the features of your test case. Previous MATLAB versions do not support test case features unavailable in that release. For example:

- Test harnesses are supported for R2015a and later.
- The Test Sequence block is supported for R2015a and later.
- `verify()` statements are supported for R2016b and later.

Test Case Limitations with Multiple Release Testing

Certain features are not supported for multiple release testing:

- Parallel test execution
- Running test cases with the MATLAB Unit Test framework
- Real-time tests
- Input data defined in an external Excel document
- Coverage collection in the Test Manager
- Generating additional tests using Simulink Design Verifier to increase coverage
- Including custom figures from test case callbacks

Add Releases Using Test Manager Preferences

Use a Test Manager preference to add to the list of release to run tests in. You can delete a release that you added to the list. You cannot delete the release from which you are running Test Manager.

1. In the Test Manager toolstrip, click **Preferences**.
2. In the **Preferences** dialog box, click **Release**. The **Release** pane lists the release you are running Test Manager from.
3. In the **Release** pane, click **Add**.
4. Browse to the location of the MATLAB release you want to add and click **OK**.

Run Baseline Tests in Multiple Releases

When you run a baseline test with Test Manager set up for multiple releases, you can:

- Create the baseline in the release you want to see the results in, for example, to try different parameters and apply tolerances.
- Create the baseline in one release and run it in another release. Using this approach you can, for example, know whether a newer release produces the same simulation outputs as an earlier release.

Create the baseline.

1. Make sure that the release has been added to your Test Manager preferences.

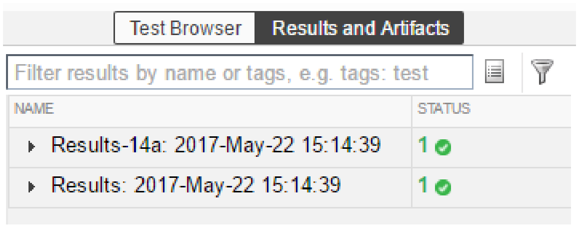
2 Create a test file, if necessary, and add a baseline test case to it.
3 In the test case, from the **Select release for simulation** list, select the releases you want to run the test case in.
4 Under **System Under Test**, enter the name of the model you want to test.
5 Set up the rest of the test.
6 Capture the baseline. Under **Baseline Criteria**, click **Capture**.
7 Select the release you want to use for the baseline simulation. Specify the file format and save and name the baseline.

For more information about capturing baselines, see “Capture Baseline Criteria” on page 7-102.

After you create the baseline, you can run the test in a release available in the Test Manager. Each release you select generates a set of results.

1. In the test case, set **Select releases for simulation** to the releases you want to use to compare against your baseline. For example, select only the release for which you created the baseline to perform a baseline comparison against the same release.
2. Specify the test options.
3. From the toolstrip, click **Run**.

   For each release that you select when you run the test case, pass-fail results appear in the **Results and Artifacts** pane. For results from a release other than the one you are running Test Manager from, the release number appears in the name.



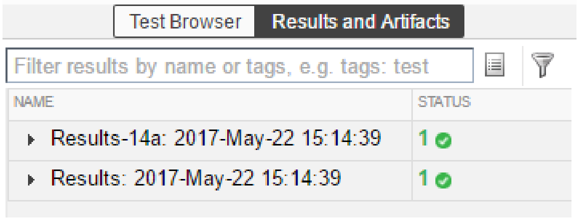
NAME	STATUS
Results-14a: 2017-May-22 15:14:39	1 ✓
Results: 2017-May-22 15:14:39	1 ✓

Run Equivalence Tests in Multiple Releases

When you run an equivalence test, you compare two simulations from the same release to see if differences in the simulations are within the specified tolerance.

1. Make sure that the release has been added to your Test Manager preferences.
2. Create a test file, if necessary, and add an equivalence test case to it.
3. In the test case, from the **Select release for simulation** list, select the releases you want to run the test case in.
4. Under **System Under Test**, enter the model you want to test.
5. Set the values under **Simulation 1** and **Simulation 2** to use as the basis for testing.
6. To set tolerances for the logged signals, under **Equivalence Criteria**, click **Capture**. Select the release you want to use for capturing the signals, and click **OK**. Clicking **Capture** copies the list of the signals being logged in Simulation 1. Then set the tolerances as desired.
7. In the toolstrip, click **Run**.

   The test runs for each release you selected, running the two simulations in the same release and comparing the results for equivalence. For each release that you selected when you ran the test case, pass-fail results appear in the **Results and Artifacts** pane. For results from a release other than the one you are running Test Manager from, the release number appears in the name.



NAME	STATUS
Results-14a: 2017-May-22 15:14:39	1 ✓
Results: 2017-May-22 15:14:39	1 ✓

Run Simulation Tests in Multiple Releases

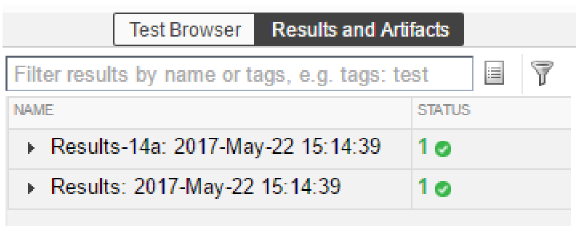
Running a simulation test simulates the model in each release you select using the criteria you specify in the test case.

1. Make sure that the release has been added to your Test Manager preferences.
2. Create a test file, if necessary, and add a simulation test case template to it.
3. In the test case, from the **Select release for simulation** list, select the releases you want to run the test case in.
4. Under **System Under Test**, enter the model you want to test.

5 Under **Simulation Outputs**, select the signals to log.

6 In the toolstrip, click **Run**.

The test runs, simulating for each release you selected. For each release, pass-fail results appear in the **Results and Artifacts** pane. For results from a release other than the one you are running Test Manager from, the release number appears in the name.



NAME	STATUS
Results-14a: 2017-May-22 15:14:39	1 ✓
Results: 2017-May-22 15:14:39	1 ✓

See Also

`sltest.testmanager.getpref` | `sltest.testmanager.setpref`

More About

- “Select Releases for Testing” on page 7-95

Examine Test Failures and Modify Baselines

After you run a baseline test in the Test Manager, you can update the baseline. For example:

- If you changed your model, you can use the new simulation output as the baseline. You can examine the failures that occurred because of the differences and update the baseline with part or all of the new output. See "Examine Test Failure Signals and Update Baseline Test" on page 7-60.
- If your test plan changed and you expect different outputs, you can manually edit the time points. See "Manually Update Signal Data in a Baseline" on page 7-63.

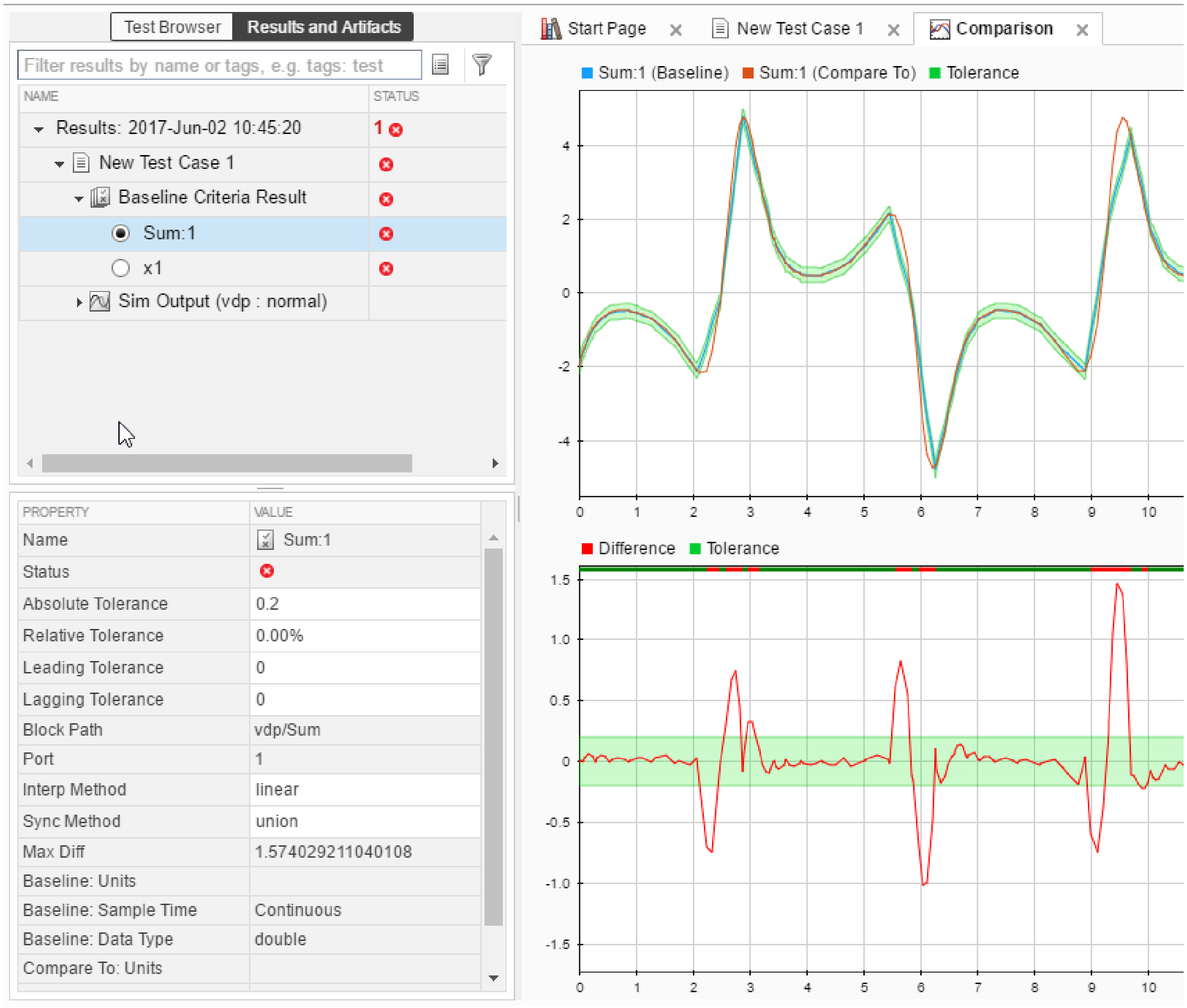
Examine Test Failure Signals and Update Baseline Test

Suppose that you run a test against a baseline and the result does not match the baseline, causing test failure. It is possible that the newer simulation better represents your desired test results or that some of the points of failure are your preferred results. You can examine the signal and failures in the data inspector view in Test Manager and decide whether you want to update the baseline or sections of the baseline.

Suppose that your model uses a new solver. When you run the test case, the results do not match, causing the test to fail.

1. Open the test file that contains the baseline test case you want to run.
2. Select the test case and run it.
3. If the test fails, in the **Results and Artifacts** pane, expand the Baseline Criteria. Select a signal that failed that you want to examine.

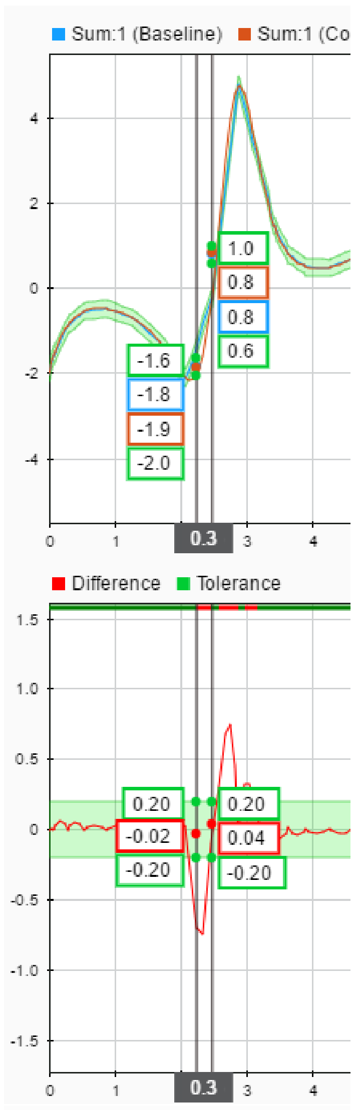
   When you select the signal, the data inspector view opens. The top graph is the baseline simulation signal overly. The bottom is the difference between those signals and the tolerance. You can adjust tolerances in the pane in the lower-left corner of the Test Manager. This example shows an absolute tolerance of `.2`.



NAME	STATUS
Results: 2017-Jun-02 10:45:20	1 ⊗
New Test Case 1	⊗
Baseline Criteria Result	⊗
● Sum:1	⊗
○ x1	⊗
Sim Output (vdp : normal)	

PROPERTY	VALUE
Name	Sum:1
Status	⊗
Absolute Tolerance	0.2
Relative Tolerance	0.00%
Leading Tolerance	0
Lagging Tolerance	0
Block Path	vdp/Sum
Port	1
Interp Method	linear
Sync Method	union
Max Diff	1.574029211040108
Baseline: Units	
Baseline: Sample Time	Continuous
Baseline: Data Type	double
Compare To: Units	

4 To examine each failure, in the toolstrip, click **Next Failure** or **Previous Failure**. Each contiguous set of failed signal comparison points makes up one region. Data cursors show the bounds of each region.



5 You can update the baseline data to use newer simulation results using the **Update Baseline** menu.

- To update the entire signal, select **Update Baseline Signal**.

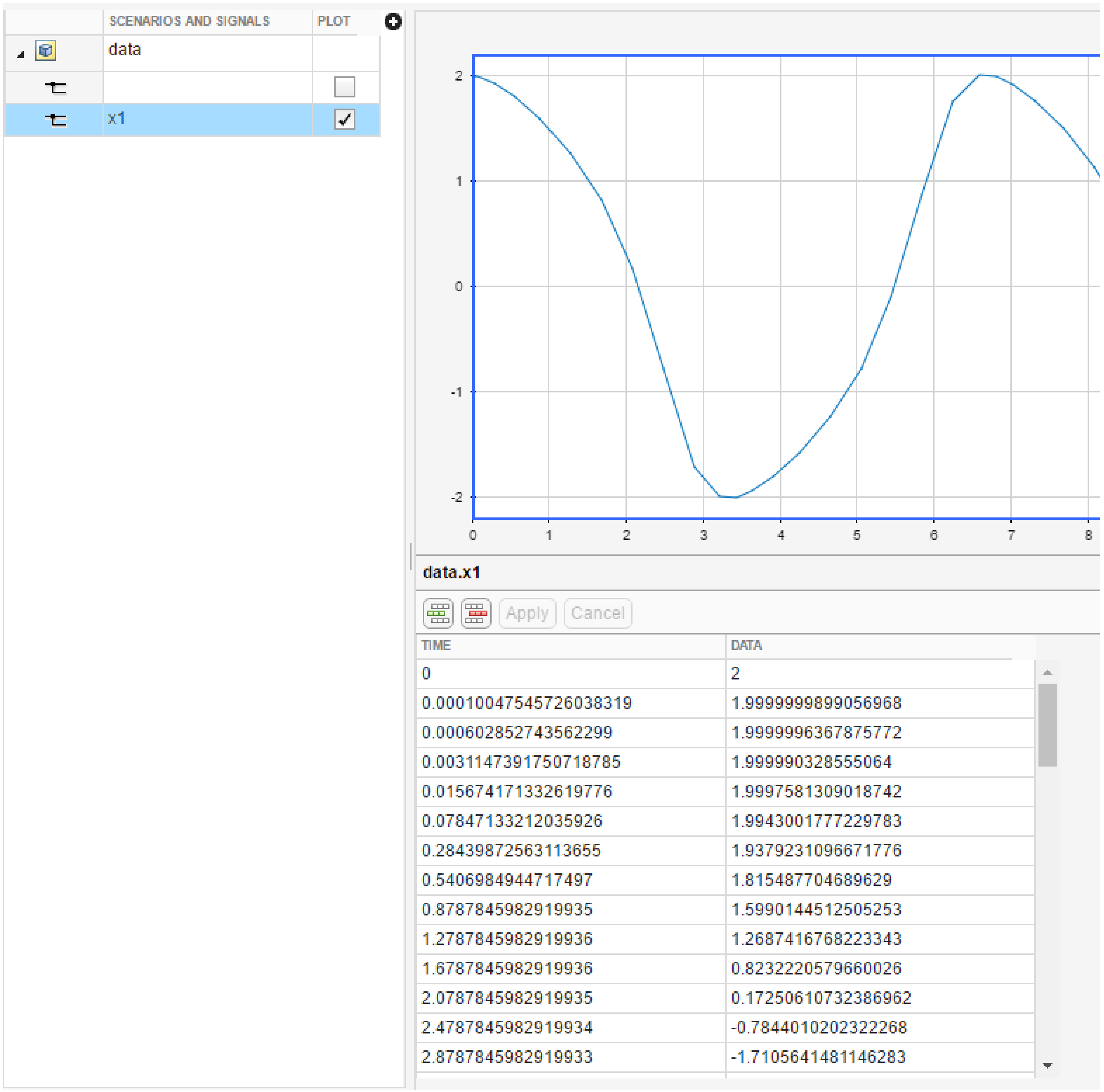
- To update only the data in the failure region, select **Update Selected Signal Region**.
- To replace all the signal data in the baseline with the new data, select **Update All Signals**.

Manually Update Signal Data in a Baseline

If your model changes such that you expect a different simulation output, you can update all or part of the baseline signal data. If the baseline is a MAT-file, you can edit the data in the signal editor. Microsoft Excel files open in Excel.

To update signal data in a MAT-file baseline:

1. Open the test file that contains the baseline you want to edit.
2. Select the test case.
3. Under **Baseline Criteria**, select the baseline whose signal data you want to edit. Click **Edit**.
4. The signal editor opens. In the **Scenarios and Signals** pane, expand the `data` node.
5. Select the check box next to the signal whose data you want to edit.



TIME	DATA
0	2
0.00010047545726038319	1.9999999899056968
0.000602852743562299	1.999999636787577 2
0.0031147391750718785	1.999990328555064
0.015674171332619776	1.9997581309018742
0.07847133212035926	1.9943001777229783
0.28439872563113655	1.937923109667177 6
0.5406984944717497	1.815487704689629
0.8787845982919935	1.5990144512505253
1.2787845982919936	1.2687416768223343
1.6787845982919936	0.8232220579660026
2.0787845982919935	0.17250610732386962
2.4787845982919934	-0.7844010202322268
2.8787845982919933	-1.710564148114628 3

Tip To see the time and data for points, display a data cursor and drag it along the signal.

6. Edit the signal data in the table, and then click **Apply**.
7. To update the baseline with the new expected output data, click **Save**.

See Also

More About

- “Work with Basic Signal Data” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Compare Model Output To Baseline Data” on page 7-9

Create and Run Test Cases with Scripts

In this section...

For a list of functions and objects in the Simulink Test programmatic interface, see “Test Scripts”.

Create and Run a Baseline Test Case

This example shows how to use `sltest.testmanager` functions, classes, and methods to automate tests and generate reports. You can create a test case, edit the test case criteria, run the test case, and generate results reports programmatically. The example compares the simulation output of the model to a baseline.

```
% Create the test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('API Test File');
ts = createTestSuite(tf,'API Test Suite');
tc = createTestCase(ts,'baseline','Baseline API Test Case');

% Remove the default test suite
tsDel = getTestSuiteByName(tf,'New Test Suite 1');
remove(tsDel);

% Assign the system under test to the test case
setProperty(tc,'Model','sldemo_absbrake');

% Capture the baseline criteria
baseline = captureBaselineCriteria(tc,'baseline_API.mat',true);

% Test a new model parameter by overriding it in the test case
% parameter set
ps = addParameterSet(tc,'Name','API Parameter Set');
po = addParameterOverride(ps,'m',55);

% Set the baseline criteria tolerance for one signal
sc = getSignalCriteria(baseline);
```

```
sc(1).AbsTol = 9;

% Run the test case and return an object with results data
ResultsObj = run(tc);

% Open the Test Manager so you can view the simulation
% output and comparison data
sltest.testmanager.view;

% Generate a report from the results data
filePath = 'test_report.pdf';
sltest.testmanager.report(ResultsObj,filePath,...
          'Author','Test Engineer',...
          'IncludeSimulationSignalPlots',true,...
          'IncludeComparisonSignalPlots',true);
```

The test case fails because only one of the signal comparisons between the simulation output and the baseline criteria is within tolerance. The results report is a PDF and opens when it is completed. For more report generation settings, see the `sltest.testmanager.report` function reference page.

Create and Run an Equivalence Test Case

This example compares signal data between two simulations to test for equivalence.

```
% Create the test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('API Test File');
ts = createTestSuite(tf,'API Test Suite');
tc = createTestCase(ts,'equivalence','Equivalence Test Case');

% Remove the default test suite
tsDel = getTestSuiteByName(tf,'New Test Suite 1');
remove(tsDel);

% Assign the system under test to the test case
% for Simulation 1 and Simulation 2
setProperty(tc,'Model','sldemo_absbrake','SimulationIndex',1);
setProperty(tc,'Model','sldemo_absbrake','SimulationIndex',2);

% Add a parameter override to Simulation 1 and 2
ps1 = addParameterSet(tc,'Name','Parameter Set 1','SimulationIndex',1);
po1 = addParameterOverride(ps1,'Rr',1.20);

ps2 = addParameterSet(tc,'Name','Parameter Set 2','SimulationIndex',2);
```

```
po2 = addParameterOverride(ps2,'Rr',1.24);

% Capture equivalence criteria
eq = captureEquivalenceCriteria(tc);

% Set the equivalence criteria tolerance for one signal
sc = getSignalCriteria(eq);
sc(1).AbsTol = 2.2;

% Run the test case and return an object with results data
ResultsObj = run(tc);

% Open the Test Manager so you can view the simulation
% output and comparison data
sltest.testmanager.view;
```

In the Equivalence Criteria Result section of the Test Manager results, the `yout.Ww` signal passes because of the tolerance value. The other signal comparisons do not pass, and the overall test case fails.

Run a Test Case and Collect Coverage

This example shows how to use a simulation test case to collect coverage results. To collect coverage, you need a Simulink Coverage license.

```
% Create the test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('API Test File');
ts = createTestSuite(tf,'API Test Suite');
tc = createTestCase(ts,'simulation','Coverage Test Case');

% Remove the default test suite
tsDel = getTestSuiteByName(tf,'New Test Suite 1');
remove(tsDel);

% Assign the system under test to the test case
setProperty(tc,'Model','sldemo_autotrans');

% Turn on coverage settings at test-file level
cov = getCoverageSettings(tf);
cov.RecordCoverage = true;

% Enable MCDC and signal range coverage metrics
cov.MetricSettings = 'mr';
```

```
% Run the test case and return an object with results data
ro = run(tf);

% Get the coverage results
tfr = getTestFileResults(ro);
tsr = getTestSuiteResults(tfr);
tcs = getTestCaseResults(tsr);
cr = getCoverageResults(tcs);

% Open the Test Manager to view results
sltest.testmanager.view;
```

In the **Results and Artifacts** pane of the Test Manager, you can view the coverage results in the test case result.

Create and Run Test Case Iterations

This example shows how to create test iterations. You can create table iterations programmatically that appear in the **Iterations** section of a test case. The example creates a simulation test case and assigns a Signal Builder group for each iteration.

```
% Create test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('Iterations Test File');
ts = getTestSuites(tf);
tc = createTestCase(ts,'simulation','Simulation Iterations');

% Specify model as system under test
setProperty(tc,'Model','sldemo_autotrans');

% Set up table iteration
% Create iteration object
testItr1 = sltestiteration;
% Set iteration settings
setTestParam(testItr1,'SignalBuilderGroup','Passing Maneuver');
% Add the iteration to test case
addIteration(tc,testItr1);

% Set up another table iteration
% Create iteration object
testItr2 = sltestiteration;
% Set iteration settings
setTestParam(testItr2,'SignalBuilderGroup','Coasting');
% Add the iteration to test case
addIteration(tc,testItr2);
```

```
% Run test case that contains iterations
results = run(tc);

% Get iteration results
tcResults = getTestCaseResults(results);
iterResults = getIterationResults(tcResults);
```

Test Iterations

In this section...

You can run the same test case with different data or configuration sets by using test case iterations. Iterations can use different:

- Parameters.
- External inputs.
- Configuration sets.
- Signal Editor scenarios.
- Signal Builder groups.
- Baseline data.

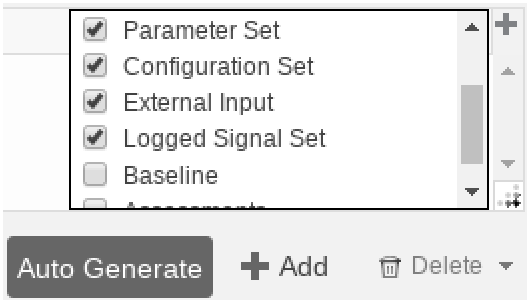
Set up iterations in the **Iterations** section of a test case. You can use table iterations or scripted iterations. If the test collects coverage using Simulink Coverage, the same coverage settings apply to all iterations in the test case.

Whether you use table or scripted iterations, you can see the iterations in the test case by clicking the **Show Iterations** button.

Create Table Iterations

Table Iterations provide a quick way to add iterations based items in your model or test case. To create iterations with the table, first make the appropriate columns visible:

1. Expand the **Iterations > Table Iterations** section.
2. In the table, add or remove columns by clicking the ➕ button and selecting items in the list. For example, to display parameter and configuration sets, select the **Parameter Set** and **Configuration Set** items.



Add Iterations Manually

1 To manually add iterations, click **Add**. The table displays a new iteration row.

2 Assign an iteration name and select items for the iteration. For example, this test case has four iterations. Each iteration uses a different combination of external input and baseline data.

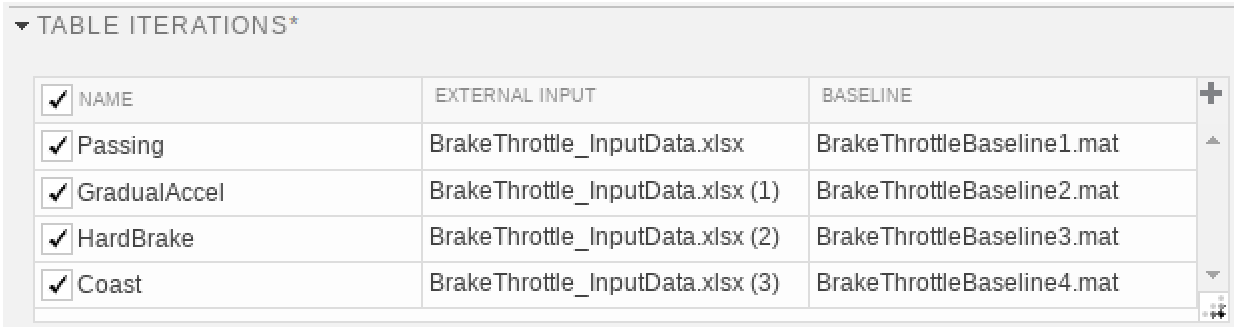


TABLE ITERATIONS*

NAME	EXTERNAL INPUT	BASELINE
✓ Passing	BrakeThrottle_InputData.xlsx	BrakeThrottleBaseline1.mat
✓ GradualAccel	BrakeThrottle_InputData.xlsx (1)	BrakeThrottleBaseline2.mat
✓ HardBrake	BrakeThrottle_InputData.xlsx (2)	BrakeThrottleBaseline3.mat
✓ Coast	BrakeThrottle_InputData.xlsx (3)	BrakeThrottleBaseline4.mat

Generate Table Iterations

You can also automatically generate iterations from data in your test case and model:

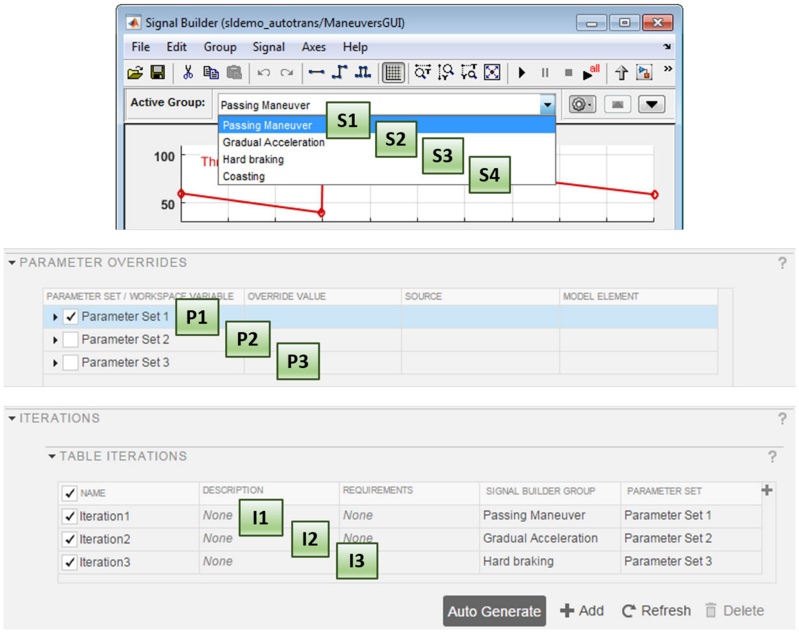
1 Click the **Auto Generate** button.

2 Select items to generate iterations.

   If you select multiple items, iterations are created in sequential pairings. For example:

   - The model `sldemo_autotrans` has a Signal Builder block with four signal groups, labeled S1, S2, S3, and S4.

- The test case has three parameter sets, labeled P1, P2, and P3.
- Automatically generating iterations from Signal Builder groups and parameter sets results in three iterations. The iterations are limited by the three parameter sets. Each iteration contains one Signal Builder group and one parameter set. The Signal Builder group and parameter set are matched in the order that they are listed in the Signal Builder block and parameter set section.



PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
✓ Parameter Set 1			
Parameter Set 2			
Parameter Set 3			

✓ NAME	DESCRIPTION	REQUIREMENTS	SIGNAL BUILDER GROUP	PARAMETER SET
✓ Iteration1	None	None	Passing Maneuver	Parameter Set 1
✓ Iteration2	None	None	Gradual Acceleration	Parameter Set 2
✓ Iteration3	None	None	Hard braking	Parameter Set 3

3 Specify an optional naming rule for the iterations. In the **Iteration naming rule** box, enter the rule using:

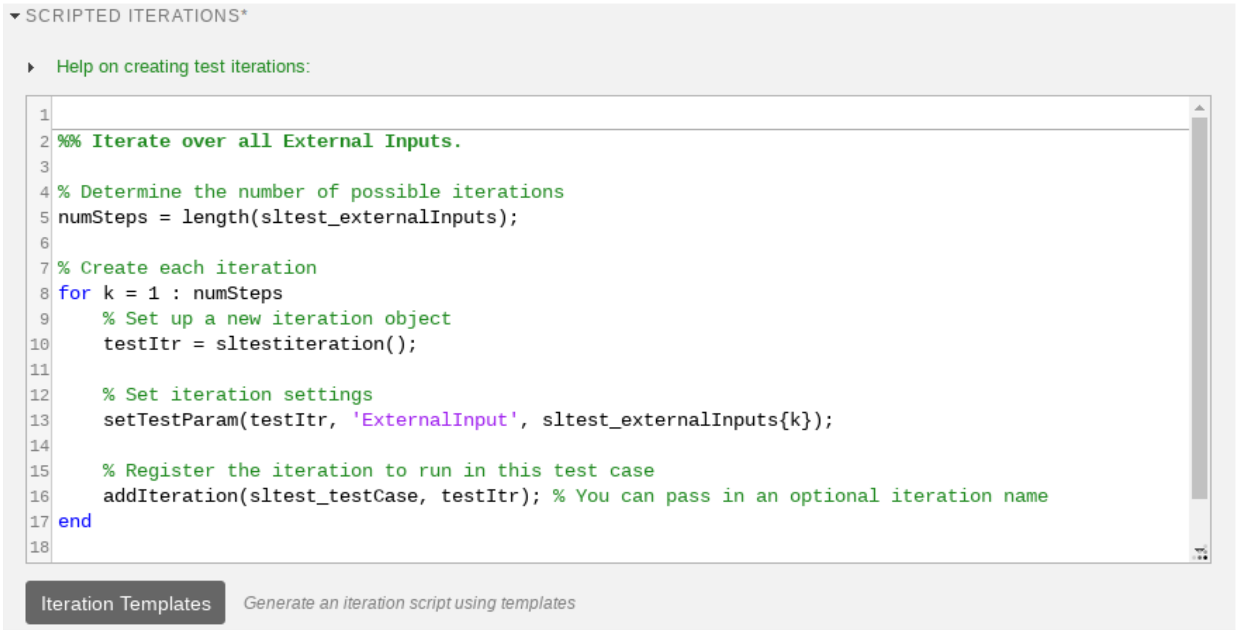
- The name of each setting you want to use in the name, with spaces removed
- An underscore or space to separate each setting

For example, if you want to include the name of the parameter set, configuration set, and baseline file name, enter `ParameterSet_ConfigurationSet_Baseline`.

Section Option	Purpose
Signal Builder Group	Applies to the **Inputs** section of a simulation, baseline, or equivalence test case, for the specified **Signal Builder Group**. Each Signal Builder group is used to generate an iteration.
Signal Editor scenario	Applies to the **Inputs** section of a simulation, baseline, or equivalence test case, for the specified **Signal Editor Scenario**. Each Signal Editor scenario is used to generate an iteration.
Parameter Set	Applies to the **Parameter Overrides** section of a simulation, baseline, or equivalence test case. Each parameter override set is used to generate an iteration.
External Input	Applies to the **Inputs** section of a simulation, baseline, or equivalence test case, for the specified **External Inputs** sets. Each external input set is used to generate an iteration.
Configuration Set	Applies to the **Configuration Setting Overrides** section of a simulation, baseline, or equivalence test case. Each iteration uses the configuration setting specified.
Baseline	Applies only to baseline test case types, specifically to the **Baseline Criteria** section of a baseline test case. Each baseline criteria set is used to generate an iteration.
Simulation 1 or 2	Applies only to equivalence test case types. At the top of the Auto Generate Reports dialog box, there is a menu for **Simulation 1** or **Simulation 2**. These sections correspond to the two simulation sections within the equivalence test case.

Create Scripted Iterations

You can run a custom set of iterations using a script in the **Scripted Iterations** section. For example, you can define parameter sets or customize iteration order by using a custom iteration. Scripted iterations are generated at run time when a test executes.


SCRIPTED ITERATIONS*
Help on creating test iterations:
Iteration Templates
Generate an iteration script using templates

```

%% Iterate over all External Inputs.

% Determine the number of possible iterations
numSteps = length(sltest_externalInputs);

% Create each iteration
for k = 1 : numSteps
    % Set up a new iteration object
    testItr = sltestiteration();

    % Set iteration settings
    setTestParam(testItr, 'ExternalInput', sltest_externalInputs{k});

    % Register the iteration to run in this test case
    addIteration(sltest_testCase, testItr); % You can pass in an optional iteration name
end

```

Iteration Script Components

An iteration script must contain certain components. The most basic iteration script contains three elements:

1. An iteration object, created using `sltestiteration`.
2. An iteration setting, set using `setTestParam`.
3. The iteration registration, added using `addIteration`.

For example, this script creates an iteration that runs one signal group from a Signal Builder block.

```
%% Iterate Using a Signal Builder Group
```

```
% Set up a new iteration object
testItr = sltestiteration;

% Set iteration setting using Signal Builder group
setTestParam(testItr,'SignalBuilderGroup',sltest_signalBuilderGroups{1});

% Add the iteration to run in this test case
% The predefined sltest_testCase variable is used here
addIteration(sltest_testCase,testItr);
```

For more information about the test iteration class, see `sltest.testmanager.TestIteration`. You can iterate over multiple items, such as Signal Builder groups. You can iterate over all Signal Builder groups in the block by putting the basic iteration script in a loop:

```
%% Iterate Over All Signal Builder Groups

% Determine the number of possible iterations
numSteps = length(sltest_signalBuilderGroups);

% Create each iteration
for k = 1 : numSteps
    % Set up a new iteration object
    testItr = sltestiteration;

    % Set iteration settings
    setTestParam(testItr,'SignalBuilderGroup',sltest_signalBuilderGroups{k});

    % Add the iteration to run in this test case
    % You can pass in an optional iteration name
    addIteration(sltest_testCase,testItr);
end
```

Predefined Variables

You can use predefined variables to write iterations scripts. To see the list of predefined variables in the Test Manager, expand the **Help on creating test iterations** section. You write the iterations script in the script box within the **Scripted Iterations** section. The script box is a functional workspace, which means the MATLAB base workspace cannot access information from the script box. If you define variables in the script box, then other workspaces cannot use the variable.

The predefined variables are:

- `sltest_bdroot` — Model simulated by the test case, defined as a string
- `sltest_sut` — The System Under Test, defined as a string
- `sltest_isharness` — `true` if `sltest_bdroot` is a harness model, defined as a logical

- `sltest_externalInputs` — Name of external inputs, defined as a cell array of strings
- `sltest_parameterSets` — Name of parameter override sets, defined as a cell array of strings
- `sltest_configSets` — Name of configuration settings, defined as a cell array of strings
- `sltest_tableIterations` — Iteration objects created in the iterations table, defined as a cell array of `sltest.testmanager.TestIteration` objects
- `sltest_testCase` — Current test case object, defined as an `sltest.testmanager.TestCase` object

Scripted Iteration Templates

You can quickly generate iterations for your test case using templates for Signal Builder groups, parameter sets, external inputs, configuration sets, and baseline sets, if you are using a baseline test case. Scripted iteration templates follow lockstep ordering and pairing of test settings. For more information about lockstep ordering, see "Create Table Iterations" on page 7-71.

For example, if you want to run all signal builder groups in a scripted iteration:

1. Click **Iteration Templates**.
2. Select the test case settings you want to iterate through. Click **OK**.

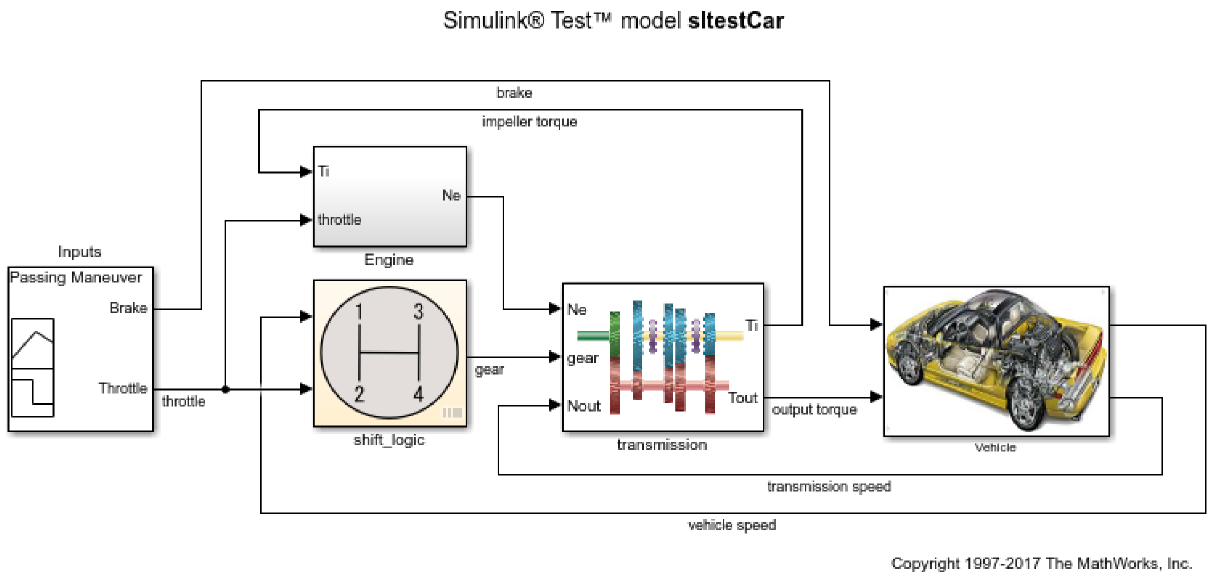
   The script is generated and added to the script box below any existing scripts.
3. To generate a table that gives a preview of the iterations that execute when you run the test case, click **Show Iterations**.

Capture Baseline Data from Iterations

This example shows how to create a baseline test by capturing data from a test case with table iterations. You create the iterations from Signal Builder groups in the model. Before running the example, navigate to a writable folder on the MATLAB® path.

1. Open the model. At the command line, enter

```
Model = 'sltestCar';
open_system(fullfile(matlabroot,'examples','simulinktest',Model));
```



2. Create a test file that contains iterations, and open the Test Manager. At the command line, enter

```
tf = sltest.testmanager.TestFile('IterationBaselineTest');
sltest.testmanager.load(tf.Name);
sltest.testmanager.view;
```

3. In the Test Manager, right-click the test case and select **Rename**. Rename the test case **Baseline Test**.

4. In the **System Under Test** section, for **Model**, enter `sltestCar`.

5. Select the signals for the baseline data:

1. In the **Simulation Outputs** section, click **Add**. The Signal Selection dialog box appears.
2. In the model canvas, select the `output torque` and `vehicle speed` signals. The signals appear in the Signal Selection dialog box.
3. In the dialog box, select both signals and click **Add**.
4. The signals appear in the **Logged Signals** table.

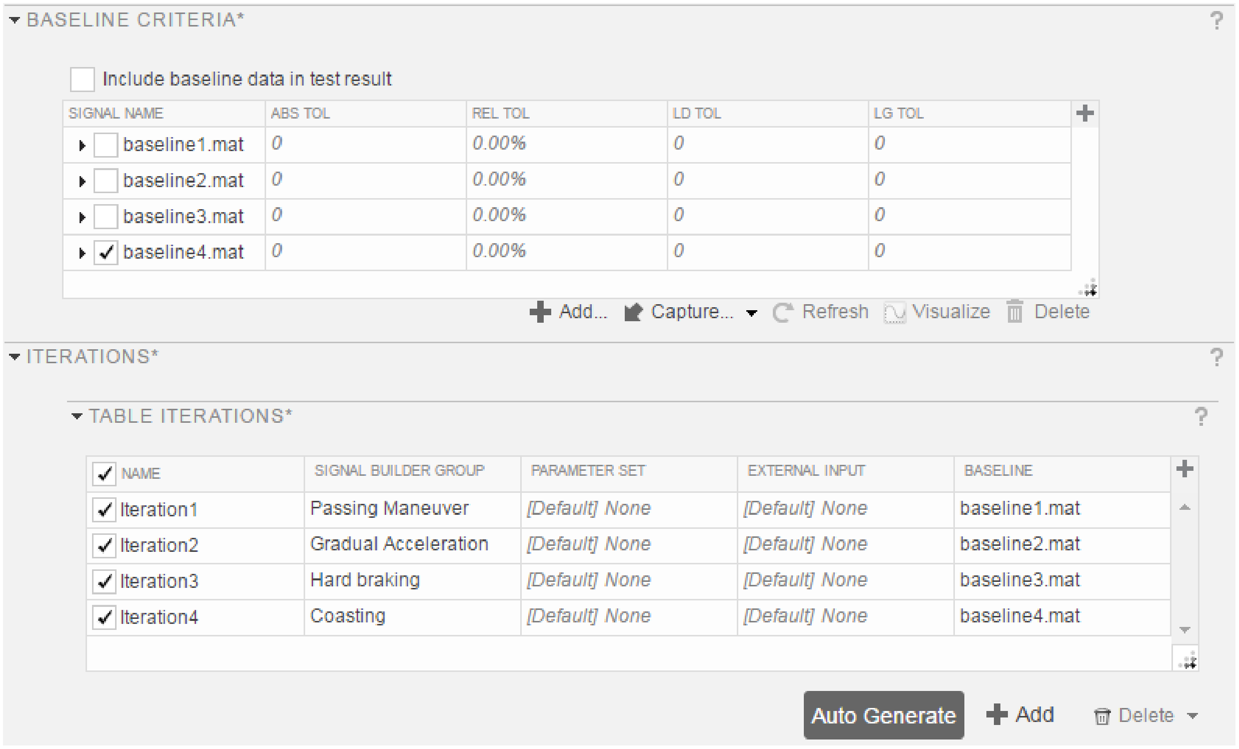
6. Add iterations for the test case:

1. Expand the **Iterations** section of the test case.
2. Expand the **Table Iterations** section and click **Auto Generate**.
3. In the dialog box, select **Signal Builder Group**. Click **OK**.
4. The table lists the iterations corresponding to the four Signal Builder groups.

7. Capture baseline data for the iterations:

1. In the **Baseline Criteria** section, click the arrow next to **Capture**, and select **Capture for Iterations**.
2. Specify a location for the baseline data files.
3. Click **Create**.

The model simulates for all Signal Builder groups. The baseline data for `output_torque` and `vehicle_speed` are captured in four MAT files. Also, each baseline data set is added to its corresponding iterations in the table.



☐ Include baseline data in test result

	SIGNAL NAME	ABS TOL	REL TOL	LD TOL	LG TOL
☐	baseline1.mat	0	0.00%	0	0
☐	baseline2.mat	0	0.00%	0	0
☐	baseline3.mat	0	0.00%	0	0
✓	baseline4.mat	0	0.00%	0	0

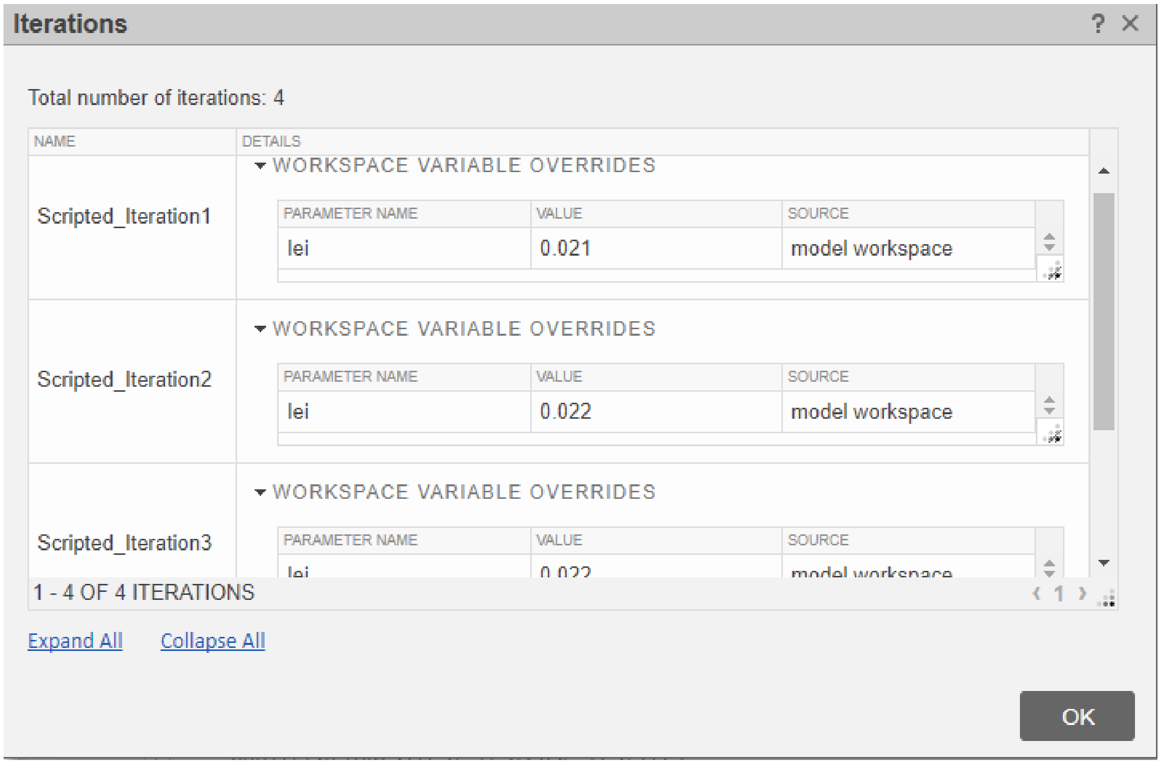
✓	NAME	SIGNAL BUILDER GROUP	PARAMETER SET	EXTERNAL INPUT	BASELINE
✓	Iteration1	Passing Maneuver	[Default] None	[Default] None	baseline1.mat
✓	Iteration2	Gradual Acceleration	[Default] None	[Default] None	baseline2.mat
✓	Iteration3	Hard braking	[Default] None	[Default] None	baseline3.mat
✓	Iteration4	Coasting	[Default] None	[Default] None	baseline4.mat

Sweep Through a Set of Parameters

Scripted iterations can be used to test a model by sweeping through a set of parameters. You can use this script to try different values for the model workspace parameter `Iei` in the model `sltestCar`. Add the script under **Iterations** > **Scripted Iterations**.

```
%% Iterate over Iei parameter

% Set up the parameter values to sweep over
IeiValues = [0.021,0.022,0.022,0.023];
numSteps = length(IeiValues);

% Create each iteration
for k = 1 : numSteps
    % Set up a new iteration object
    testItr = sltestiteration;
    % Set value of Iei (parameter in model workspace)
    setVariable(testItr,'Name','Iei','Source','model workspace',...
                'Value',IeiValues(k));

    % Add the iteration to run in this test case
    addIteration(sltest_testCase,testItr);
end
```

After you add the script, click **Show Iterations**. You can see the iterations that the script created.



Total number of iterations: 4

NAME	DETAILS
Scripted_Iteration1	WORKSPACE VARIABLE OVERRIDES
Scripted_Iteration2	WORKSPACE VARIABLE OVERRIDES
Scripted_Iteration3	WORKSPACE VARIABLE OVERRIDES

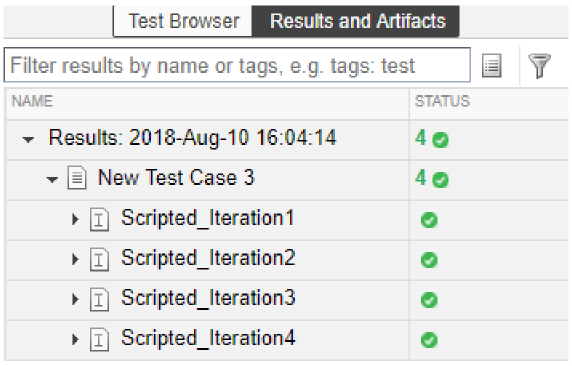
PARAMETER NAME	VALUE	SOURCE
Iei	0.021	model workspace

PARAMETER NAME	VALUE	SOURCE
Iei	0.022	model workspace

PARAMETER NAME	VALUE	SOURCE
Iei	0.022	model workspace

1 - 4 OF 4 ITERATIONS

Running the test generates a result for each iteration.



NAME	STATUS
Results: 2018-Aug-10 16:04:14	4 ✓
New Test Case 3	4 ✓
Scripted_Iteration1	✓
Scripted_Iteration2	✓
Scripted_Iteration3	✓
Scripted_Iteration4	✓

See Also

`sltest.testmanager.TestIteration`

Related Examples

- "Create and Run Test Cases with Scripts" on page 7-66

Collect Coverage in Tests

In this section...
"Enable and Collect Coverage for a Test File" on page 7-83
"Considerations for Collecting Coverage in Test Harnesses" on page 7-86

If you use Simulink Coverage to generate model and code coverage, then you can collect coverage metrics when you run your test cases. For test cases with coverage collection turned on, Test Manager includes the coverage of each metric you specify to collect in the results.

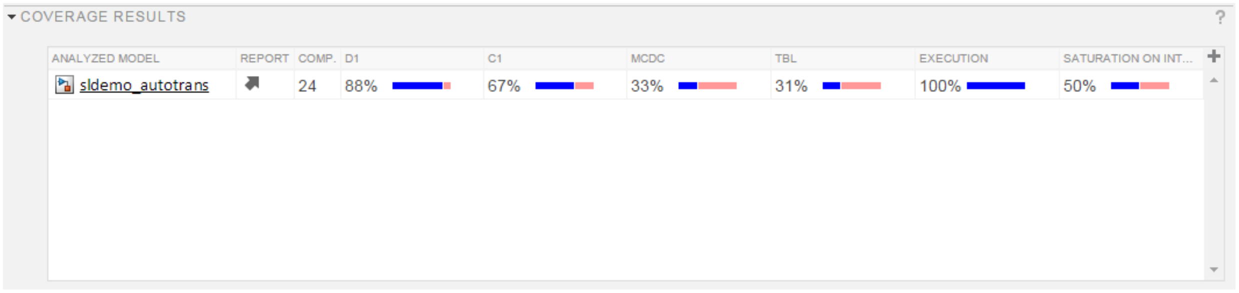
To test a model for coverage, turn on coverage collection on the test file and specify the metrics you want to collect. Test suites and test cases inherit the settings from the test file. After coverage is turned on at the file level, you can turn it off and on for each test suite or test case.

Enable and Collect Coverage for a Test File

Enable coverage collection, view coverage results in Test Manager, and trace coverage results from Test Manager to the model. The model `sldemo_autotrans` specifies coverage.

For information about types of model coverage, see "Types of Model Coverage" (Simulink Coverage).

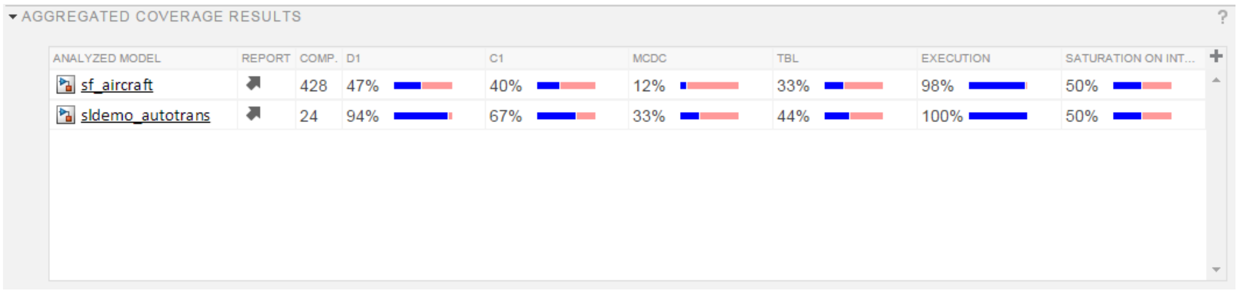
1. Create a test file and set up the test case for your model.
2. In the test file settings, under **Coverage Settings**, select **Record coverage for system under test**. You can also specify whether to collect coverage for referenced models.
3. Select the coverage metrics that you want to collect.
4. Run the test file.
5. To view the coverage results, select the test case result in the **Results and Artifacts** pane and expand the **Coverage Results** section.



COVERAGE RESULTS

ANALYZED MODEL	REPORT	COMP.	D1	C1	MCDC	TBL	EXECUTION	SATURATION ON INT...
sldemo_autotrans		24	88%	67%	33%	31%	100%	50%

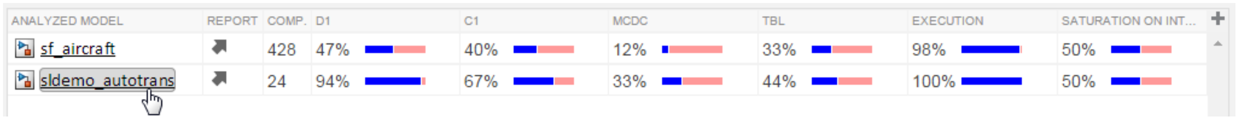
6 If your test file or test suite collected coverage metrics for more than one model, you can view all the coverage metrics in one place. Select the result and expand the **Aggregated Coverage Results** section.



AGGREGATED COVERAGE RESULTS

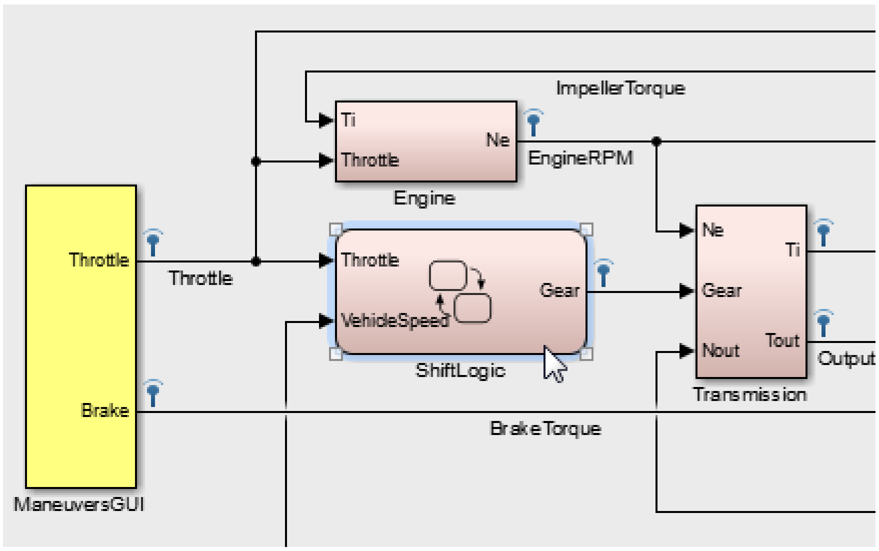
ANALYZED MODEL	REPORT	COMP.	D1	C1	MCDC	TBL	EXECUTION	SATURATION ON INT...
sf_aircraft		428	47%	40%	12%	33%	98%	50%
sldemo_autotrans		24	94%	67%	33%	44%	100%	50%

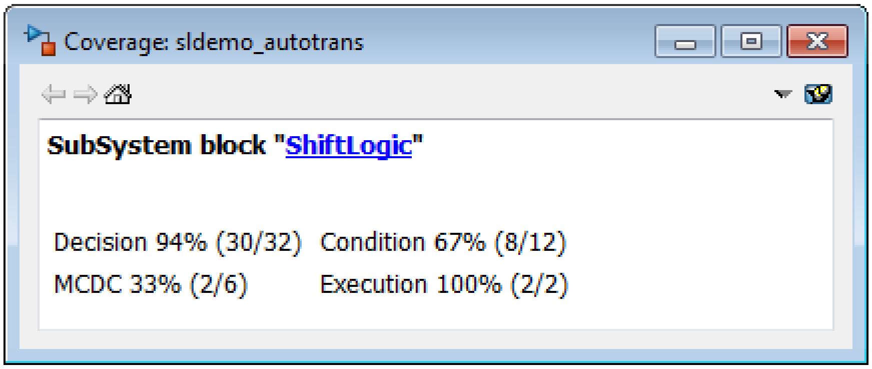
7 To trace the coverage results to the model, click the model name in the coverage results table.



ANALYZED MODEL	REPORT	COMP.	D1	C1	MCDC	TBL	EXECUTION	SATURATION ON INT...
sf_aircraft		428	47%	40%	12%	33%	98%	50%
sldemo_autotrans		24	94%	67%	33%	44%	100%	50%

In the model, select model elements to see the coverage data.

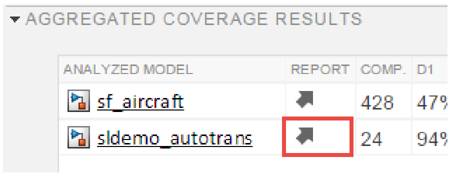




SubSystem block "ShiftLogic"

Decision 94% (30/32) Condition 67% (8/12)
MCDC 33% (2/6) Execution 100% (2/2)

8 To create a report of the coverage for a model, click the arrow under the **Report** column in the coverage results.



AGGREGATED COVERAGE RESULTS

ANALYZED MODEL	REPORT	COMP.	D1
sf_aircraft		428	47%
sldemo_autotrans		24	94%

Tip To see aggregated results from different test files, in the **Results and Artifacts** pane, select the test file results whose coverage results you want to see in the same results file. From the context menu, select **Merge Coverage Results**. A results file that contains the combined coverage results appears in the list.

Considerations for Collecting Coverage in Test Harnesses

Loading coverage results to a model, or aggregating coverage results across models, requires a model consistent with the coverage results. Therefore, to perform aggregated coverage collection, it is recommended that you use test harnesses configured to automatically synchronize the component under test. Set **SynchronizationMode** to `Synchronize on harness open and close`. For more information, see "Synchronize Changes Between Test Harness and Model" on page 2-53.

Coverage results association depends on test harness – main model synchronization:

- If the test harness is configured to synchronize the component under test when you open or close the harness, coverage results from the test harness are associated with the main model. When you close the test harness, the coverage results remain active in memory. You can aggregate coverage with additional results collected from the main model or another synchronized test harness.
- If the test harness is configured to only synchronize the component under test when you manually push or rebuild, the coverage results are associated with the test harness.

  - When you close the test harness, the coverage results are removed from memory.
  - If the component under test design differs between test harness and main model, you cannot aggregate coverage results.
  - You can aggregate coverage results with the main model if the component under test design does not differ, but you must manually load the coverage results into the main model. See the function cvload.

See Also

`sltest.testmanager.CoverageSettings`

Related Examples

- "Perform Functional Testing and Analyze Test Coverage" on page 10-11
- "Create and Run Test Cases with Scripts" on page 7-66
- "Specify Coverage Options" (Simulink Coverage)

Run Tests Using Parallel Execution

In this section...

If you have a license to Parallel Computing Toolbox, then you can execute tests in parallel using a parallel pool (parpool). Running tests in parallel can speed up execution and decrease the amount of time it takes to get test results.

When Do Tests Benefit from Using Parallel Execution?

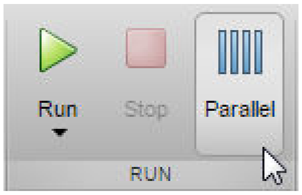
In general, parallel execution can help reduce test execution time if you have

- A complex Simulink model that takes a long time to simulate.
- Numerous long-running tests, such as iterations.

Use Parallel Execution

To run a test file using parallel execution:

1 The Test Manager uses the default Parallel Computing Toolbox cluster. For information about where to specify or change the cluster, see "Discover Clusters and Use Cluster Profiles" (Parallel Computing Toolbox). Test Manager runs in parallel only on the local machine.
2 On the Test Manager toolstrip, click the **Parallel** button.



3 Run a test file. The test file executes using parallel pool.
4 To turn off parallel execution, click the **Parallel** button to toggle it off.

Starting a parallel pool can take time, which would slow down test execution. To reduce time:

- Make sure that the parallel pool is already running before you run a test. By default, the parallel pool shuts down after being idle for a specified number of minutes. To change the setting, see “Specify Your Parallel Preferences” (Parallel Computing Toolbox).
- Load Simulink on all the parallel pool workers.

See Also

`sltest.testmanager.run`

Related Examples

- “Clusters and Clouds” (Parallel Computing Toolbox)

Set Signal Tolerances

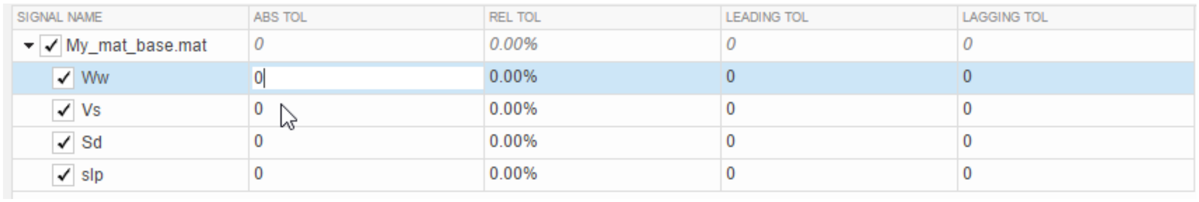
In this section...
"Modify Criteria Tolerances" on page 7-90
"Change Leading Tolerance in a Baseline Comparison Test" on page 7-90

You can specify tolerances in the **Baseline Criteria** or **Equivalence Criteria** sections of baseline and equivalence test cases. You can specify relative, absolute, leading, and lagging tolerances for a signal comparison.

To learn about how tolerances are calculated, see "How the Simulation Data Inspector Compares Data" (Simulink).

Modify Criteria Tolerances

To modify a tolerance, select the signal name in the criteria table, double-click the tolerance value, and enter a new value.



SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
✓ My_mat_base.mat	0	0.00%	0	0
✓ Ww	0	0.00%	0	0
✓ Vs	0	0.00%	0	0
✓ Sd	0	0.00%	0	0
✓ slp	0	0.00%	0	0

If you modify a tolerance after you run a test case, rerun the test case to apply the new tolerance value to the pass/fail results.

Change Leading Tolerance in a Baseline Comparison Test

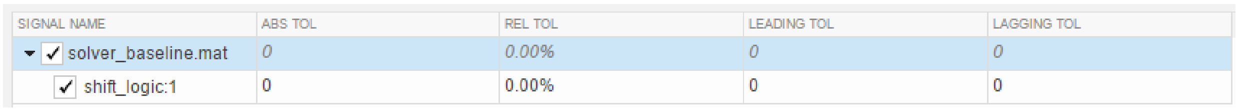
Specify a tolerance when the difference between results falls in a range you consider acceptable. Suppose that your model under test uses a particular solver. Solvers are sometimes updated from one release to the next, and new solvers also become available. If you use an updated solver or change solvers, you can specify an acceptable tolerance for differences between your baseline and later tests.

Generate the Baseline

Generate the baseline for the `sf_car` model, which uses the `ode-5` solver.

1 Open the model `sf_car`.
2 Open the Test Manager and create a test file named `Solver Compare`. In the test case, set the system under test to `sf_car`.
3 Select the signal to log. Under **Simulation Outputs**, click **Add**. In the model, select the `shift_logic` output signal. In the Signal Selection dialog box, select the check box next to `shift_logic` and click **Add**.
4 Save the baseline. Under **Baseline Criteria**, click **Capture**. Set the file format to `MAT`. Name the baseline `solver_baseline` and click **Capture**.

   After you capture the baseline MAT-file, the model runs and the baseline criteria appear in the table. Each default tolerance is 0.



SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
✓ solver_baseline.mat	0	0.00%	0	0
✓ shift_logic:1	0	0.00%	0	0

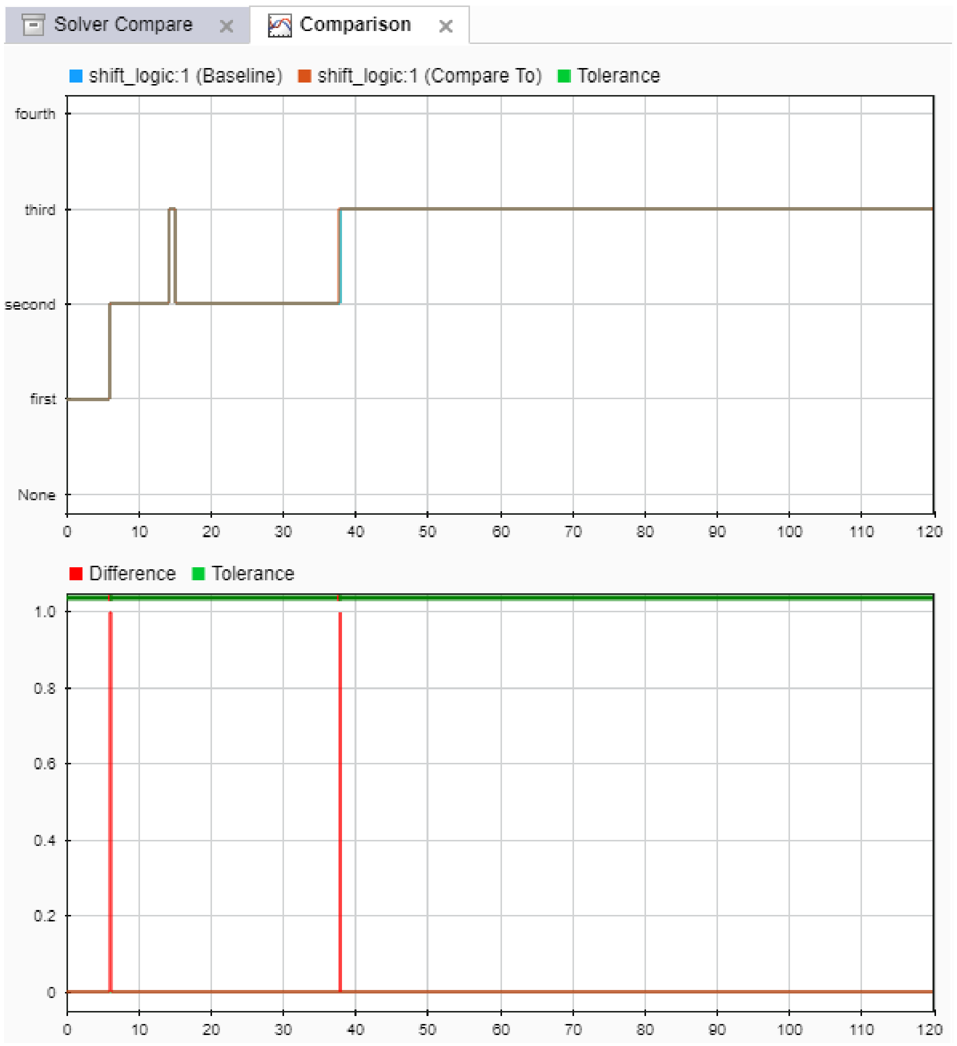
Change Solvers and Run the Test Case

Suppose that you want to use a different solver with your model. You run a test to compare results using the new solver with the baseline.

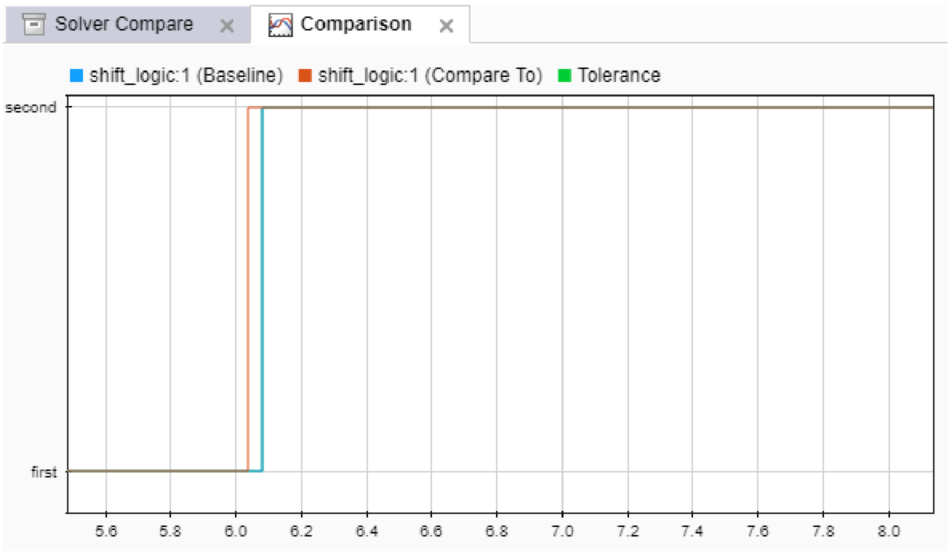
1 In the model, change the solver to `ode1`.
2 In the Test Manager, with the `Solver Compare` test file selected, click **Run**.

   In the **Results and Artifacts** pane, notice that the test failed.
3 Expand the results of the failed test. Under **Baseline Criteria Result**, select the `shift_logic` signal.

   The **Comparison** tab shows where the difference occurred.



4 Zoom the comparison chart where the results diverged. The comparison signal changes ahead of the baseline, that is, it leads the baseline signal.

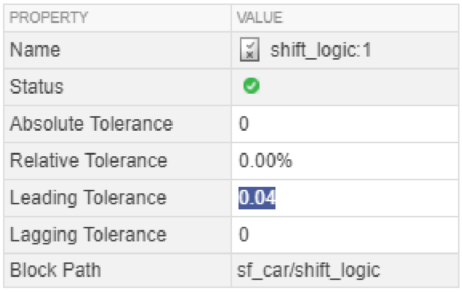


Preview and Set a Leading Tolerance Value

Suppose that your team determines that a tolerance the size of the simulation step size (.04 in this case) is acceptable. In the Test Manager, set a leading tolerance value. Use a leading tolerance for the signal whose change occurs ahead of your baseline. Use a lagging tolerance for a signal whose change occurs after your baseline.

You can preview how the tolerance value affects the test to see if the test passes with the specified tolerance. Then set the tolerance on the baseline criteria and rerun the test.

1 Preview whether the tolerance you want to use causes the test to pass. With the result signal selected, in the property box, set **Leading Tolerance** to `.04`.



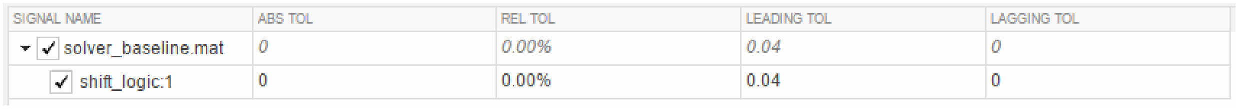
PROPERTY	VALUE
Name	shift_logic:1
Status	✓
Absolute Tolerance	0
Relative Tolerance	0.00%
Leading Tolerance	0.04
Lagging Tolerance	0
Block Path	sf_car/shift_logic

When you change this value, the status changes to show that the failed tests pass.

2 When you are satisfied with the tolerance value, enter it in the baseline criteria so you can rerun the test and save the new pass-fail result. In the **Test Browser** pane, select the test case in the `Solver Compare` test.

3 Under **Baseline Criteria**, change the **Leading Tol** value for the `solver_baseline.mat` file to `.04`.

By default, each signal inherits this value from the baseline file. You can override the value for each signal.



SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
✓ solver_baseline.mat	*0*	*0.00%*	*0.04*	*0*
✓ shift_logic:1	0	0.00%	0.04	0

4 Run the test again. The test passes.

5 To store the tolerance value and the passed test with the test file, save the test file.

See Also

`sltest.testmanager.BaselineCriteria` | `sltest.testmanager.SignalCriteria`

Related Examples

- “Compare Model Output To Baseline Data” on page 7-9

Test Sections

In this section...

To view or edit the test sections, select a test file, suite, or case in the **Test Browser** pane.

Select Releases for Testing

You can select MATLAB releases installed on your system to create and run tests in. Use this preference to specify the MATLAB installations that you want to make available for testing with Test Manager. You can use releases from R2011b forward. The releases you add become available to select from the **Select releases for simulation** list when you design the test.

You can add releases to the list and delete them. You cannot delete the release you started MATLAB in.

To add a release, click **Add**, navigate to the location of the MATLAB installation you want to add, and click **OK**.

For more information, see “Run Tests in Multiple Releases” on page 7-55.

Set Preferences to Display Test Sections

To simplify the Test Manager layout, you can select the sections of the test case, test suite, or test file that appear in the Test Manager. Test case sections that were modified appear in the Test Manager, regardless of the preference setting.

1. In the toolstrip, click **Preferences**.
2. Select the **Test File**, **Test Suite**, or **Test Case** tab.
3. Select sections to show, or clear sections to hide. To show only sections where settings are set, clear all selections in the **Preferences** dialog box.
4. Click **OK**.

Also see `sltest.testmanager.getpref` and `sltest.testmanager.setpref`.

Select releases for simulation

Select the releases that you want available for running test cases. Build the list of releases using the **Release** pane in the Test Manager Preferences dialog box. For more information, see “Run Tests in Multiple Releases” on page 7-55.

Tags

Tag your tests with useful categorizations, such as `safety`, `logged-data`, or `burn-in`. Filter tests using these tags when executing tests or viewing results. See “Filter Test Execution and Results” on page 7-135.

Description

In this section, add descriptive text to your test case, test suite, or test file.

Requirements

If you have a Simulink Requirements license, you can establish traceability by linking your test cases to requirements. For more information, see “Link to Test Cases from Requirements” (Simulink Requirements).

To link a test case, test suite, or test file to a requirement:

1. Open the Requirements Editor. In the Simulink menu, select **Analysis** > **Requirements** > **Requirements Editor**.
2. Highlight a requirement.
3. In the Test Manager, in the **Requirements** section, click the arrow next to the **Add** button and select **Link to Selected Requirement**.
4. The requirement link appears in the **Requirements** list.

System Under Test

Specify the model you want to test in the **System Under Test** section. To use an open model in the currently active Simulink window, click the **Use current model** button.

Note The model must be available on the path to run the test case. You can add the model's containing folder to the path using the preload callback. See “Callbacks” on page 7-99.

Specifying a new model in the **System Under Test** section can cause the model information to be out of date. To update the model test harnesses, Signal Builder groups, and available configuration sets, click the **Refresh** button.

Test Harness

If you have a test harness in your system under test, then you can select the test harness to use for the test case. If you have added or removed test harnesses in the model, click the **Refresh** button to view the updated test harness list.

For more information about using test harnesses, see “Refine, Test, and Debug a Subsystem” on page 2-20.

Simulation Settings

You can override the **System Under Test** simulation settings such as the simulation mode, start time, stop time, and initial state.

Considerations

- The **System Under Test** cannot be in fast restart or external mode.
- To stop a test running in **Rapid Accelerator** mode, press **Ctrl+C** at the MATLAB command prompt.
- When running parallel execution in rapid accelerator mode, streamed signals do not show up in the Test Manager.
- The **System Under Test** cannot be a protected model.

Parameter Overrides

In this section, you can specify parameter values in the test case to override the parameter values in the model workspace, data dictionary, or base workspace. Parameters are grouped into sets. You can turn parameter sets and individual parameter overrides on or off by using the check box next to the set or parameter.

To add a parameter override:

1. Click **Add**.

   A dialog box opens with a list of parameters. If the list of parameters is not current, click the **Refresh** button ↻ in the dialog box.
2. Select the parameter you want to override.
3. To add the parameter to the parameter set, click **OK**.
4. Enter the override value in the parameter **Override Value** column.

To restore the default value of a parameter, clear the value in the **Override Value** column and press **Enter**.

You can also add a set of parameter overrides from a MAT-file. Click the **Add** arrow and select `Add File` to create a parameter set from a MAT-file.

For an example that uses parameter overrides, see “Override Model Parameters in a Test Case”.

Considerations

The Test Manager displays only top-level system parameters from the system under test.

Callbacks

Test-File Level Callbacks

Two callback scripts are available in each test suite that execute at different times during a test:

- Setup runs before test file executes.
- Cleanup runs after test file executes.

Test-Suite Level Callbacks

Two callback scripts are available in each test suite that execute at different times during a test:

- Setup runs before the test suite executes.
- Cleanup runs after the test suite executes.

Test-Case Level Callbacks

Three callback scripts are available in each test case that execute at different times during a test:

- Pre-load runs before the model loads and before the model callbacks.
- Post-load runs after the model loads and the `PostLoadFcn` model callback.
- Cleanup runs after simulations and model callbacks.

To run a single callback script, click the **Run** button ▶ above the corresponding script.

You can use predefined variables in the test case callbacks:

- `sltest_bdroot` available in **Post-Load**: The model simulated by the test case. The model can be a harness model.
- `sltest_sut` available in **Post-Load**: The system under test. For a harness, it is the component under test.

- `sltest_isharness` available in **Post-Load**: Returns true if `sltest_bdroot` is a harness model.
- `sltest_simout` available in **Cleanup**: Simulation output produced by simulation.
- `sltest_iterationName` available in **Pre-Load**, **Post-Load**, and **Cleanup**: Name of the currently executing test iteration.

The test case callback scripts are not stored with the model and do not override Simulink model callbacks. Consider the following when using callbacks:

- To stop execution of an infinite loop from a callback script, press **Ctrl+C** at the MATLAB command prompt.
- `sltest.testmanager` functions are not supported.

Inputs

A test case can use input data from:

- A Signal Builder or Signal Editor block in the system under test. Select **Signal Editor scenario or Signal Builder group**, and select the scenario or signal group. The system under test can have only one Signal Builder or Signal Editor block at the top level.
- An external data file. In the **External Inputs** table, click **Add**. Select a MAT-file or Microsoft Excel file.

  For more information on using external files as inputs, see “Run Tests Using External Data” on page 7-25. For information about the file format for Microsoft Excel files in Test Manager, see “Format Test Case Data in Excel” on page 7-38.
- An input file template that you create and populate with data. See “Test Case Input Data Files” on page 7-34.

To include the input data in your test results set, select **Include input data in test result**.

If the time interval of your input data is shorter than the model simulation time, you can limit the simulation to the time specified by your input data by selecting **Stop simulation at last time point**.

For more information on test inputs, see the Test Authoring: Inputs page.

Edit Input Data Files in Test Manager

From the Test Manager, you can edit your input data files.

To edit a file, select the file and click **Edit**. You can then edit the data in the signal editor for MAT-files or Microsoft Excel for Excel files.

To learn about the syntax for Excel files, see “Format Test Case Data in Excel” on page 7-38.

Simulation Outputs

Use the **Simulation Outputs** section to add signal outputs to your test results. Signals logged in your model or test harness can appear in the results after you add them as simulation outputs. You can then plot them. Add individual signals to log and plot or add a signal set.

Under **Simulation Outputs**, click **Add**. Follow the user interface. For more information, see “Capture Simulation Data in a Test Case” on page 7-49.

Configuration Setting Overrides

In the test case, you can specify configuration settings that differ from the settings in the model. Setting the configuration settings in the test case enables you to try different configurations without modifying your model.

Simulation 1 and Simulation 2

These sections appear in equivalence test cases. Use them to specify the details about the simulations that you want to compare. Enter the system under test, the test harness if applicable, and simulation setting overrides under **Simulation 1**. You can then click **Copy settings from Simulation 1** under **Simulation 2** to use a starting point for your second set of simulation settings.

For the test to pass, Simulation 1 and Simulation 2 must log the same signals.

Use these sections with the **Equivalence Criteria** section to define the premise of your test case. For an example of an equivalence test, see “Test Two Simulations for Equivalence”.

Equivalence Criteria

This section appears in equivalence test cases. The equivalence criteria is a set of signal data to compare in Simulation 1 and Simulation 2. Specify tolerances to regulate pass-fail criteria of the test. You can specify absolute, relative, leading, and lagging tolerances for the signals.

To specify tolerances, first click **Capture** to run the system under test in Simulation 1 and add signals marked for logging to the table. Specify the tolerances in the table.

After you capture the signals, you can select signals from the table to narrow your results. If you do not select signals under **Equivalence Criteria**, running the test case compares all the logged signals in Simulation 1 and Simulation 2.

For an example of an equivalence test case, see “Test Two Simulations for Equivalence”.

Baseline Criteria

The **Baseline Criteria** section appears in baseline test cases. When a baseline test case executes, Test Manager captures signal data from signals in the model marked for logging and compares them to the baseline data.

Capture Baseline Criteria

To capture logged signal data from the system under test to use as the baseline criteria, click **Capture**. Then follow the prompts in the Capture Baseline dialog box. Capturing the data compiles and simulates the system under test and stores the output from the logged signals to the baseline. For a baseline test example, see “Compare Model Output To Baseline Data” on page 7-9.

You can save the signal data to a MAT-file or a Microsoft Excel file. To understand the format of the Excel file, see “Format Test Case Data in Excel” on page 7-38.

You can capture the baseline criteria using the current release for simulation or another release installed on your system. Add the releases you want to use in the Test Manager preferences. Then, select the releases you want available in your test case using the **Select releases for simulation** option in the test case. When you run the test, you can compare the baseline against the release you created the baseline in or against another release. For more information, see “Run Tests in Multiple Releases” on page 7-55.

When you select Excel as the output format, you can specify the sheet name to save the data to. If you use the same Excel file for input and output data, by default both sets of data appear in the same sheet.

If you are capturing the data to a file that already contains outputs, specify the sheet name to overwrite the output data only in that sheet of the file.

To save a baseline for each test case iteration in a separate sheet in the same file, select **Capture a baseline for each iterations**. This check box appears only if your test case already contains iterations. For more information iterations, see “Test Iterations” on page 7-71.

Specify Tolerances

You can specify tolerances to determine the pass-fail criteria of the test case. You can specify absolute, relative, leading, and lagging tolerances for individual signals or the entire baseline criteria set.

After you capture the baseline, the baseline file and its signals appear in the table. In the table, you can set the tolerances for the signals. To see tolerances used in an example for baseline testing, see “Compare Model Output To Baseline Data” on page 7-9.

Add File as Baseline

By clicking **Add**, you can select an existing file as a baseline. You can add MAT-files and Microsoft Excel files as the baseline. Format Microsoft Excel files as described in “Format Test Case Data in Excel” on page 7-38.

Update Signal Data in Baseline

You can edit the signal data in your baseline, for example, if your model changed and you expect different values. To open the signal editor or the Microsoft Excel file for editing, select the baseline file from the list and click **Edit**. See “Manually Update Signal Data in a Baseline” on page 7-63.

You can also update your baseline when you examine test failures in the data inspector view. See “Examine Test Failures and Modify Baselines” on page 7-60.

Logical and Temporal Assessments

Create temporal assessments using the form-based editor that prompts you for conditions, events, signal values, delays, and responses. When you collapse the individual elements, the editor displays a readable statement summarizing the assessment. See "Assess Temporal Logic Using Temporal Assessments" on page 3-82.

Custom Criteria

This section includes an embedded MATLAB editor to define custom pass/fail criteria for your test. Select **function customCriteria(test)** to enable the criteria script in the editor. Custom criteria operate outside of model run time; the script evaluates after model simulation.

Common uses of custom criteria include verifying signal characteristics or verifying test conditions. MATLAB Unit Test qualifications provide a framework for verification criteria. For example, this custom criteria script gets the last value of the signal `PhiRef` and verifies that it equals `0`:

```
% Get the last value of PhiRef from the dataset Signals_Req1_3
lastValue = test.sltest_simout.get('Signals_Req1_3').get('PhiRef').Values.Data(end);

% Verify that the last value equals 0
test.verifyEqual(lastValue,0);
```

See "Process Test Results with Custom Scripts" on page 7-111. For a list of MATLAB Unit Test qualifications, see "Types of Qualifications" (MATLAB).

You can also define plots in the **Custom Criteria** section. See "Create, Store, and Open MATLAB Figures" on page 7-122.

Iterations

Use iterations to repeat a test with different parameter values, configuration sets, or input data.

- You can run multiple simulations with the same inputs, outputs, and criteria by sweeping through different parameter values in a test case.
- Models and external data files can contain multiple test input scenarios, such as signal groups. To simplify your test file architecture, you can run different input scenarios as iterations rather than as different test cases. You can apply different baseline data to each iteration, or capture new baseline data from an iteration set.
- You can iterate over different configuration sets, for example to compare results between solvers or data types.

To create iterations from defined parameter sets, signal groups, external data files, or configuration sets, use table iterations. To create a custom set of iterations from the available test case elements, write a MATLAB iteration script in the test case. For more information about test iterations, see “Test Iterations” on page 7-71

Coverage Settings

Use this test section to configure coverage collection for test files, test suites, and test cases. For more information about collecting coverage in your test, see “Collect Coverage in Tests” on page 7-83.

Test File Options

Close open Figures at the end of execution

When your tests generate figures, select this option to clear the working environment of figures after the test execution completes.

Store MATLAB figures

Select this option to store figures generated during the test with the test file. You can enter MATLAB code that creates figures and plots as a callback or in the test case **Custom Criteria** section. See “Create, Store, and Open MATLAB Figures” on page 7-122.

Generate report after execution

Select **Generate report after execution** to create a report after the test executes. Selecting this option displays report options that you can set. The settings are saved with the test file.

For detailed reporting information, see “Export Test Results and Generate Reports” on page 8-9 and “Customize Test Reports” on page 8-15.

See Also

`sltest.testmanager.getpref` | `sltest.testmanager.setpref`

Increase Coverage by Generating Test Inputs

In this section...

Using Simulink Design Verifier, you can generate test inputs that replicate design errors, achieve test objectives, or meet coverage criteria. Simulink Test can create test cases that use test inputs and expected outputs from Simulink Design Verifier.

Overall Workflow

Test case generation follows this workflow.

1. Choose an existing Simulink Design Verifier results file, or generate new results by analyzing your model.

   - If you use an existing results file, you can load results by either:

     - Using the Simulink Test command `sltest.import.sldvData`.
     - Using Simulink Design Verifier menu items. In the model, select **Analysis** > **Design Verifier** > **Results** > **Load**. Select the MAT file with the analysis results.
   - If you run a model analysis, the Simulink Design Verifier Results Summary window appears after the analysis completes.
2. In the results summary window, click **Export test cases to Simulink Test**.
3. Enter the name of an existing or new test harness.
4. Select a test harness source for the generated test inputs. You can select

   - `Inport`: The inputs are contained in the Simulink Design Verifier data file and mapped to Inport blocks in the test harness. The mapping is shown in the **Inputs** section of the test case. Using the `Inport` option allows you to map other inputs to the test harness Inport blocks, which can be useful for running multiple test cases or iterations using the same test harness.
   - `Signal Builder`: The inputs are contained in groups in a Signal Builder block inside the test harness. Using the `Signal Builder` option allows you to view the test inputs in the Signal Builder block editor.

5. Select a new or existing test file, and enter names for the test file and test case.
6. Click OK to export the test cases to Simulink Test. The test files and test cases are updated in the Test Manager.

Test Case Generation Example

This example shows how to generate test cases for a controller subsystem using Simulink Design Verifier, and export the test cases to a test file in Simulink Test. The example requires a Simulink Design Verifier license.

The model is a closed-loop heat pump system. The controller accepts the measured room temperature and set temperature inputs. The controller outputs a bus of three signals controlling the fan, heat pump, and the direction of the heat pump (heat or cool). The model contains a harness that tests heating and cooling scenarios.

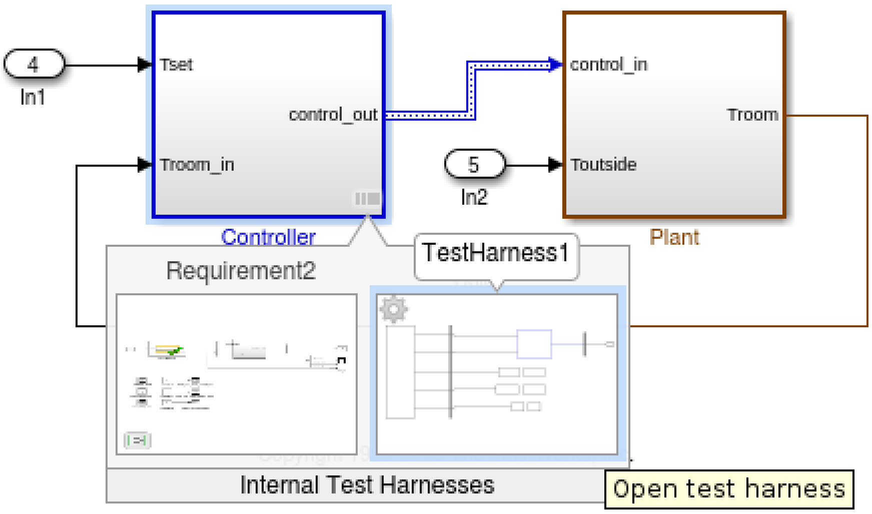
1. Open the model.

   ```
   open_system(fullfile(docroot,'toolbox','sltest','examples',...
   'sltestTestCaseFromDVExample.slx'));
   ```
2. Set the current working folder to a writable folder.
3. In the model, generate tests for the `Controller` subsystem. Right-click the `Controller` block and select **Design Verifier** > **Generate Tests for Subsystem**.

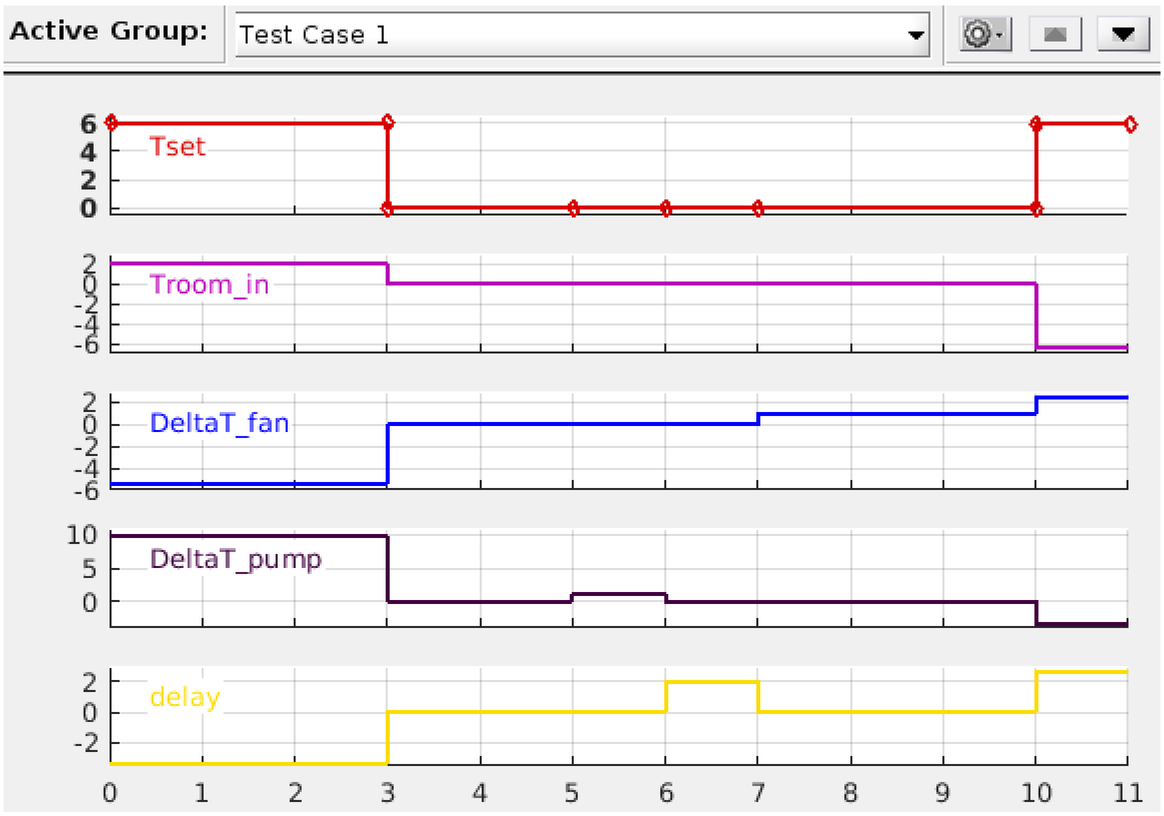
   Simulink Design Verifier generates tests for the component.
4. In the results summary window, click **Export test cases to Simulink Test**.
5. In the Export Design Verifier Test Cases dialog box, enter:

   - Test Harness: `TestHarness1`
   - Harness Source: `Signal Builder`
   - Select **Use a new test file**
   - Test File: `./TestFile_GeneratedTests.mldatx`
   - Test Case: `<Create a new test case>`
6. Click **OK**.

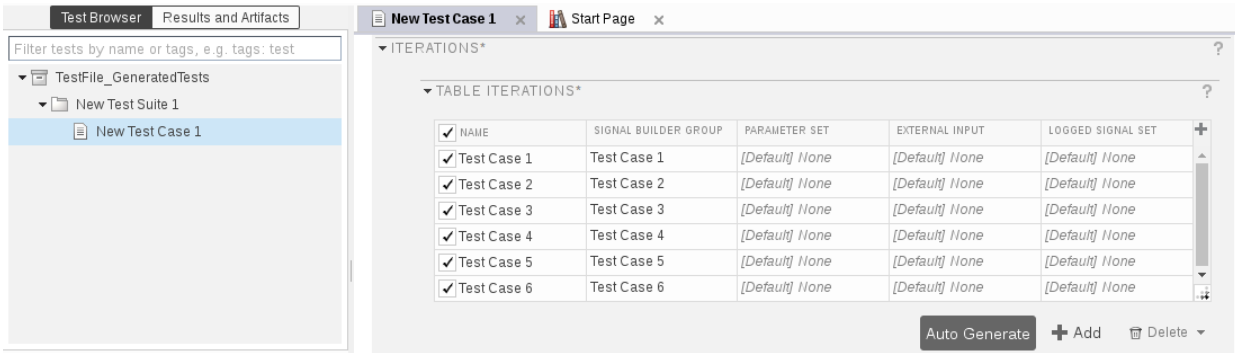
   A new test file is created in the working folder, and a test harness is added to the main model, owned by the `Controller` subsystem. Click the harness badge to preview the new test harness.



7 Click the `TestHarness1` thumbnail to open the harness, and double-click the Signal Builder block source to see the generated inputs.



8 In the Test Manager, the new test case displays the system under test, and the test harness containing the generated inputs in the Signal Builder source. Expand the **Iterations** section to see the iterations corresponding to the signal builder groups.



✓ NAME	SIGNAL BUILDER GROUP	PARAMETER SET	EXTERNAL INPUT	LOGGED SIGNAL SET
✓ Test Case 1	Test Case 1	[Default] None	[Default] None	[Default] None
✓ Test Case 2	Test Case 2	[Default] None	[Default] None	[Default] None
✓ Test Case 3	Test Case 3	[Default] None	[Default] None	[Default] None
✓ Test Case 4	Test Case 4	[Default] None	[Default] None	[Default] None
✓ Test Case 5	Test Case 5	[Default] None	[Default] None	[Default] None
✓ Test Case 6	Test Case 6	[Default] None	[Default] None	[Default] None

See Also

`sltest.import.sldvData`

Process Test Results with Custom Scripts

In this section...

Testing your model often requires assessing conditions that ensure a test is valid, in addition to verifying model behavior. MATLAB Unit Test provides a framework for such assessments. In Simulink Test, you can use the test case custom criteria to author specific assessments, and include MATLAB Unit Test qualifications in your script.

Custom criteria apply as post-simulation criteria to the simulation output. If you require run-time verifications, use a `verify()` statement in a Test Assessment or Test Sequence block. See "Assess Model Simulation Using verify Statements" on page 3-15.

MATLAB Testing Framework

A custom criteria script is a method of `test`, which is a `matlab.unittest` test case object. To enable the function, in the test case **Custom Criteria** section of the Test Manager, select **function customCriteria(test)**. Inside the function, enter the custom criteria script in the embedded MATLAB editor.

The embedded MATLAB editor lists properties of `test`. Create test assessments using MATLAB Unit Test qualifications. Custom criteria supports verification and assertion type qualifications. See "Types of Qualifications" (MATLAB). Verifications and assertions operate differently when custom criteria are evaluated:

- Verifications – Other assessments are evaluated when verifications fail. Diagnostics appear in the results. Use verifications for general assessments, such as checking simulation against expected outputs.

  Example: `test.verifyEqual(lastValue,0)`
- Assertions – The custom criteria script stops evaluating when an assertion fails. Diagnostics appear in the results. Use assertions for conditions that render the criteria invalid.

Example: `test.assertEqual(lastValue,0)`.

Define a Custom Criteria Script

This example shows how to create a custom criteria script for an autopilot test case.

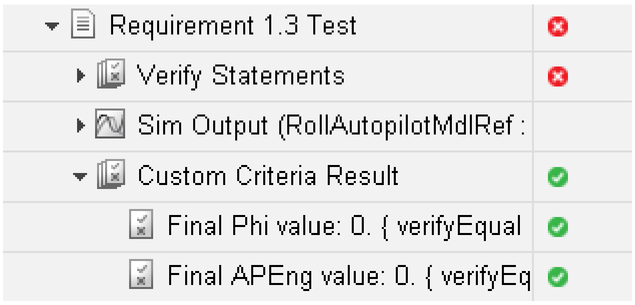
1. Open the test file.

   ```
   sltest.testmanager.load('AutopilotTestFile.mldatx')
   sltest.testmanager.view
   ```

2. In the **Test Browser**, select **AutopilotTestFile** > **Basic Design Test Cases** > **Requirement 1.3 Test**. In the test case, expand the **Custom Criteria** section.
3. Enable the custom criteria script by selecting `function customCriteria(test)`.
4. In the embedded MATLAB editor, enter the following script. The script gets the final value of the signals `Phi` and `APEng`, and verifies that the final values equal `0`.

   ```
   % Get the last values
   lastPhi = test.sltest_simout.get('Signals_Req1_3').get('Phi').Values.Data(end);
   lastAPEng = test.sltest_simout.get('Signals_Req1_3').get('APEng').Values.Data(end);

   % Verify the last values equal 0
   test.verifyEqual(lastPhi,0,['Final Phi value: ',num2str(lastPhi),'.']);
   test.verifyEqual(lastAPEng,false,['Final APEng value: ',num2str(lastAPEng),'.']);
   ```

5. Run the test case.
6. In the **Results and Artifacts** pane, expand the **Custom Criteria** Result. Both criteria pass.


Requirement 1.3 Test ✗
Verify Statements ✗
Sim Output (RollAutopilotMdlRef :
Custom Criteria Result ✓
Final Phi value: 0. { verifyEqual ✓
Final APEng value: 0. { verifyEq ✓

Reuse Custom Criteria and Debug Using Breakpoints

In addition to authoring criteria scripts in the embedded MATLAB editor, you can author custom criteria in a standalone function, and call the function from the test case. Using a standalone function allows you

- To reuse the custom criteria in multiple test cases.
- To set breakpoints in the criteria script for debugging.
- To investigate the simulation output using the command line.

In this example, you add a breakpoint to a custom criteria script. You run the test case, list the properties of the test object at the command line, and call the custom criteria from the test case.

Call Custom Criteria Script from the Test Case

1 Navigate to the folder containing the criteria function.

```
cd(fullfile(docroot,'toolbox','sltest','examples'))
```

2 Open the custom criteria script

```
open('sltestCheckFinalRollRefValues.m')
```

```
% This is a custom criteria function for a Simulink Test test case.
% The function gets the last values of Phi and APEng from the
% Requirements 1.3 test case in the test file AutopilotTestFile.

function sltestCheckFinalRollRefValues(test)

% Get the last values
lastPhi = test.sltest_simout.get('Signals_Req1_3').get('Phi').Values.Data(end)
lastAPEng = test.sltest_simout.get('Signals_Req1_3').get('APEng').Values.Data(end)

% Verify the last values equal 0
test.verifyEqual(lastPhi,0,['Final Phi value: ',num2str(lastPhi),'.']);
test.verifyEqual(lastAPEng,false,['Final APEng value: ',num2str(lastAPEng),'.']);
```

3 Open the test file

```
sltest.testmanager.load('AutopilotTestFile.mldatx')
sltest.testmanager.view
```

4 In the embedded MATLAB editor under **Custom Criteria**, enter the function call to the custom criteria:

```
sltestCheckFinalRollRefValues(test)
```

Set Breakpoints and List test Properties

1. On line 8 of `sltestCheckFinalRollRefValues.m`, set a breakpoint by clicking the dash to the right of the line number.
2. In the Test Manager, run the test case.

   The command window displays a debugging prompt.
3. Enter `test` at the command prompt to display the properties of the `STMCustomCriteria` object. The properties contain characteristics and simulation data output of the test case.

   ```
   test = 

     STMCustomCriteria with properties:

               TestResult: [1×1 sltest.testmanager.TestCaseResult]
                   sltest_simout: [1×1 Simulink.SimulationOutput]
          sltest_testCase: [1×1 sltest.testmanager.TestCase]
            sltest_bdroot: {'RollReference_Requirement1_3'}
               sltest_sut: {'RollAutopilotMdlRef/Roll Reference'}
         sltest_isharness: 1
     sltest_iterationName: ''
   ```

   The property `sltest_simout` contains the simulation data. To view the data `PhiRef`, enter

   ```
   test.sltest_simout.get('Signals_Req1_3').get('PhiRef')
   ```

   ```
   ans = 

     Simulink.SimulationData.Signal
     Package: Simulink.SimulationData

     Properties:
     struct with fields:

               Name: 'PhiRef'
     PropagatedName: ''
          BlockPath: [1×1 Simulink.SimulationData.BlockPath]
           PortType: 'outport'
          PortIndex: 1
             Values: [1×1 timeseries]
   ```

4 In the MATLAB editor, click **Continue** to continue running the custom criteria script.

5 In the **Results and Artifacts** pane, expand the **Custom Criteria** Result. Both criteria pass.

6 To reuse the script in another test case, call the function from the test case custom criteria.

Assess the Damping Ratio of a Flutter Suppression System

Using a custom criteria script, verify that wing oscillations are damped in multiple altitude and airspeed conditions.

The Simulation and Model

The model uses Simscape™ to simulate a Benchmark Active Controls Technology (BACT) / Pitch and Plunge Apparatus (PAPA) setup. It uses Aerospace Blockset™ to simulate aerodynamic forces on the wing.

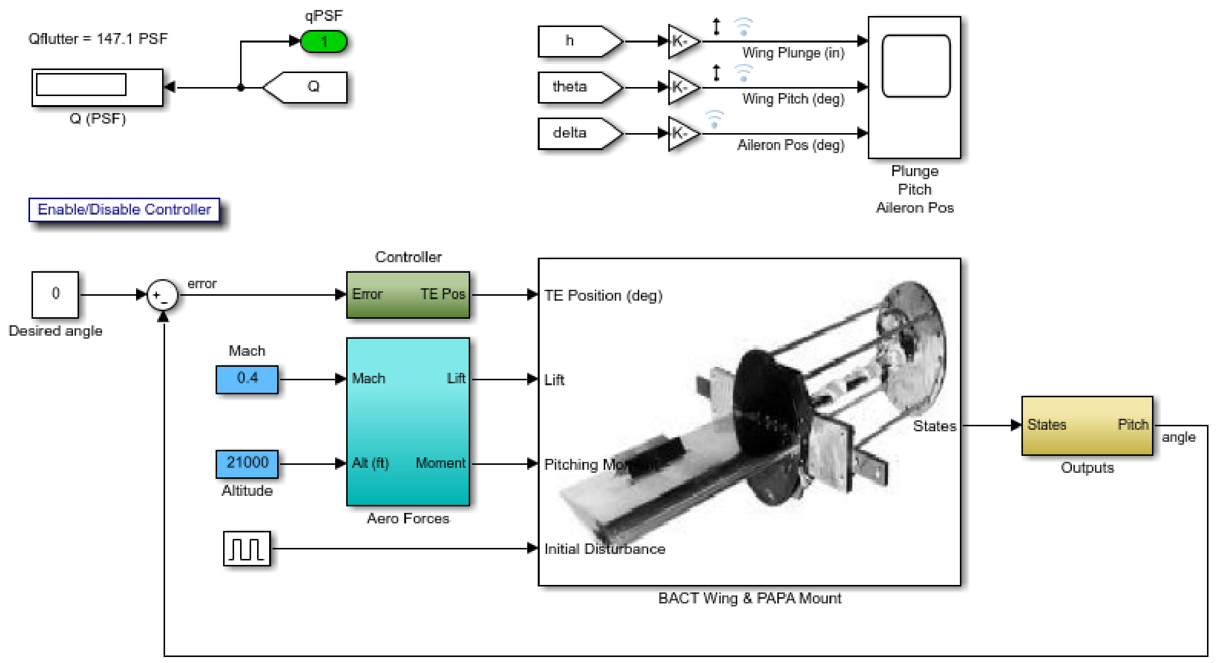
The test iterates over 16 combinations of `Mach` and `Altitude`. The test case uses custom criteria with Curve Fitting Toolbox™ to find the peaks of the wing pitch, and determine the damping ratio. If the damping ratio is not greater than zero, the assessment fails.

Running this test case requires

- Simulink® Test™
- Simscape Multibody™
- Aerospace Blockset™
- Curve Fitting Toolbox™

Open the model and the test file.

```
open_system(fullfile(matlabroot,'examples','simulinktest',...
    'sltestFlutterSuppressionSystemExample.slx'))
```



```
open(fullfile(matlabroot,'examples','simulinktest',...
    'sltestFlutterCriteriaTest.mldatx'))
```

Custom Criteria Script

The test case custom criteria uses this script to verify that the damping ratio is greater than zero.

```
% Get time and data for pitch
Time = test.sltest_simout.get('sigsOut').get('pitch').Values.Time(1:15000);
Data = test.sltest_simout.get('sigsOut').get('pitch').Values.Data(1:15000);

% Find peaks
[~, peakIds] = findpeaks(Data,'minpeakheight', 0.002, 'minpeakdistance', 50);
peakTime= Time(peakIds);
peakPos = Data(peakIds);
rn = peakPos(1)./peakPos(2:end);
L = 1:length(rn);
```

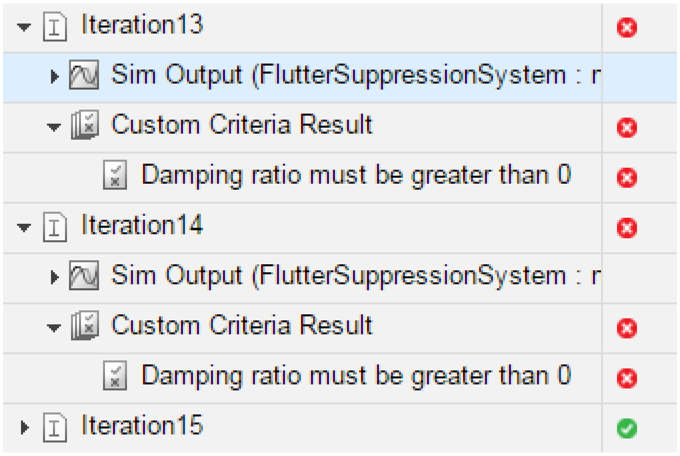
```
% Do curve fitting
fittedModel = exponentialFitAndPlot(L, rn);
delta = fittedModel.d;

% Find damping ratio
dRatio = delta/sqrt((2*pi)^2+delta^2);

% Make sure damping ratio is greater than 0
test.verifyGreaterThan(dRatio,0,'Damping ratio must be greater than 0');
```

Test Results

Running the test case returns two conditions in which the damping ratio is greater than zero.

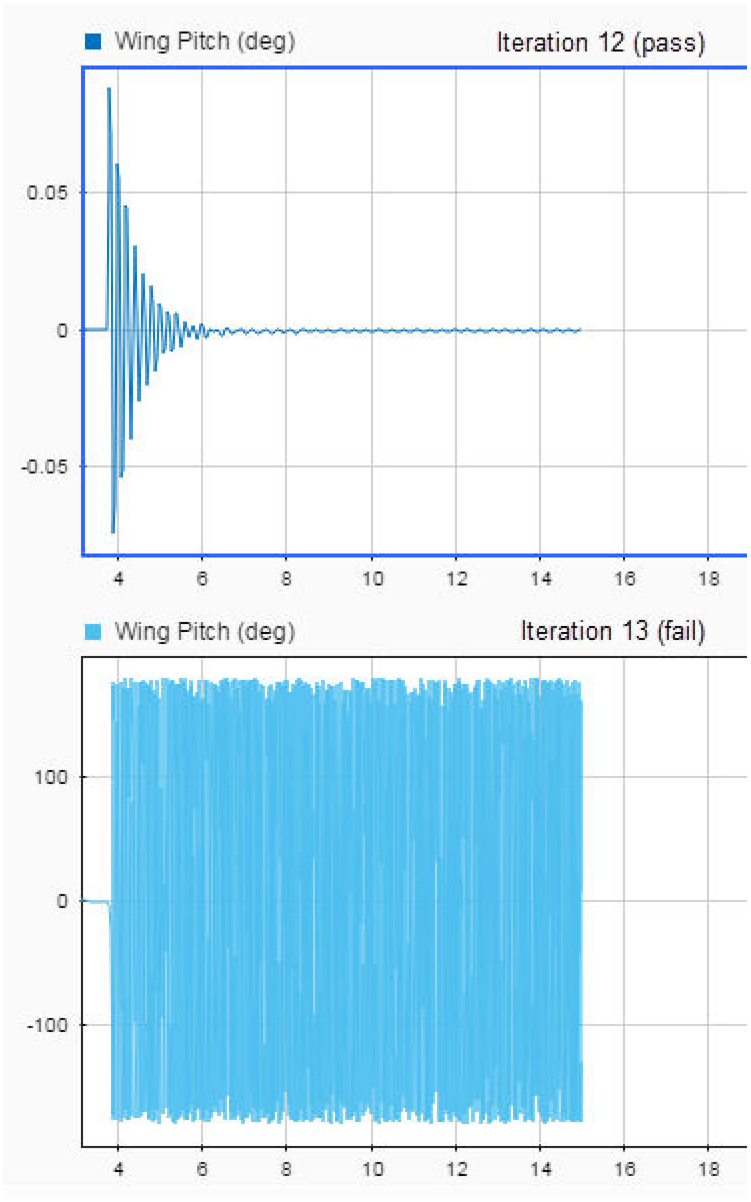
```
results = sltest.testmanager.run
```

```
results = 

  ResultSet with properties:

                   Name: 'Results: 2019-Mar-03 16:55:16'
              NumPassed: 14
              NumFailed: 2
            NumDisabled: 0
          NumIncomplete: 0
               NumTotal: 16
     NumTestCaseResults: 0
    NumTestSuiteResults: 0
     NumTestFileResults: 1
                Outcome: Failed
              StartTime: 03-Mar-2019 16:55:19
               StopTime: 03-Mar-2019 16:58:59
               Duration: 220 sec
        CoverageResults: []
                Release: ''
```



Iteration13 ✖
Sim Output (FlutterSuppressionSystem : r
Custom Criteria Result ✖
Damping ratio must be greater than 0 ✖
Iteration14 ✖
Sim Output (FlutterSuppressionSystem : r
Custom Criteria Result ✖
Damping ratio must be greater than 0 ✖
Iteration15 ✓

The wing pitch plots from iteration 12 and 13 show the difference between a positive damping ratio (iteration 12) and a negative damping ratio (iteration 13).



```
sltest.testmanager.close
```

```
close_system('sltestFlutterSuppressionSystemExample.slx',0)
```

Custom Criteria Programmatic Interface Example

This example shows how to set and get custom criteria using the programmatic interface.

Before running this example, temporarily disable warnings that result from verification failures.

```
warning off Stateflow:Runtime:TestVerificationFailed;
warning off Stateflow:cdr:VerifyDangerousComparison;
```

Load a Test File and Get Test Case Object

```
tf = sltest.testmanager.load('AutopilotTestFile.mldatx');

ts = getTestSuiteByName(tf,'Basic Design Test Cases');

tc = getTestCaseByName(ts,'Requirement 1.3 Test');
```

Create the Custom Criteria Object and Set Criteria

Create the custom criteria object.

```
tcCriteria = getCustomCriteria(tc)
```

```
tcCriteria = 
  CustomCriteria with properties:

     Enabled: 0
    Callback: '% Return value: customCriteria...'
```

Create the custom criteria expression. This script gets the last value of the signal `Phi` and verifies that it equals `0`.

```
criteria = ...
    sprintf(['lastPhi = test.SimOut.get(''Signals_Req1_3'')',...
    '.get(''Phi'').Values.Data(end);\n',...
    'test.verifyEqual(lastPhi,0,[''Final: '',num2str(lastPhi),''.'']);'])
```

```
criteria = 
    'lastPhi = test.SimOut.get('Signals_Req1_3').get('Phi').Values.Data(end);
     test.verifyEqual(lastPhi,0,['Final: ',num2str(lastPhi),'.']);'
```

Set and enable the criteria.

```
tcCriteria.Callback = criteria;
tcCriteria.Enabled = true;
```

Run the Test Case and Get the Results

Run the test case.

```
tcResultSet = run(tc);
```

Get the test case results.

```
tcResult = getTestCaseResults(tcResultSet);
```

Get the custom criteria result.

```
ccResult = getCustomCriteriaResult(tcResult)
```

```
ccResult = 
  CustomCriteriaResult with properties:

             Outcome: Failed
    DiagnosticRecord: [1x1 sltest.testmanager.DiagnosticRecord]
```

Restore warnings from verification failures.

```
warning on Stateflow:Runtime:TestVerificationFailed;
warning on Stateflow:cdr:VerifyDangerousComparison;

sltest.testmanager.clearResults
sltest.testmanager.clear
sltest.testmanager.close
```

See Also

Related Examples

- “Test Models Using MATLAB Unit Test” on page 7-125
- “Create, Store, and Open MATLAB Figures” on page 7-122

Create, Store, and Open MATLAB Figures

In this section...
"Create a Custom Figure for a Test Case" on page 7-122
"Include Figures in a Report" on page 7-124

You can create figures using MATLAB commands to include with test results and reports. Enter the commands in a test case section that accepts MATLAB code. These sections include the test case **Custom Criteria** section, and callbacks that can execute with your test case.

If you include code that creates figures with your test case, you can:

- Display the figures after the test runs
- Store the figures with your test case
- Include them in a report
- Access stored figures from your test results

To specify this behavior, use the **Test File Options** section under the **Test File** settings.

- Select **Close all open figures at the end of execution** if you do not need to see the figures right after the test executes, for example, if you are storing the figures or including them in a report. Clear this check box if you are not storing the figures and you want to view them after the test executes.
- Select **Store MATLAB figures** if you want to save the figures with the test results. This option also enables you to open the figures from the results and to include them in a report.

After you run the test, the figures appear under **MATLAB Figures** in the test case results.

Create a Custom Figure for a Test Case

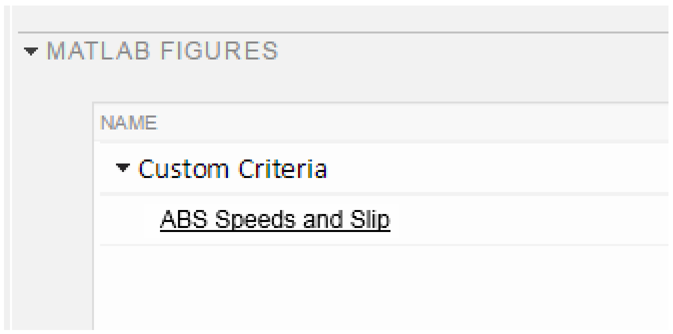
In this example, add code that creates a figure to the **Custom Criteria** section of a test case. To access the figure from the test results, set options on the test file.

1. Open the model `sldemo_absbrake`.
2. In the Test Manager, create a test file and name it `custom_figures`.

3. In the default test case, under **System Under Test**, set the model to `sldemo_absbrake`.
4. Under **Custom Criteria**, select the **function customCriteria(test)** check box and paste this code in the text box.

```
h = findobj(0,'Name','ABS Speeds and Slip');
if isempty(h)
    h=figure('Position',[26   100   452   700],...
        'Name','ABS Speeds and Slip',...
        'NumberTitle','off');
end
figure(h)
set(h,'DefaultAxesFontSize',8)

% Log data in sldemo_absbrake_output
out = test.sltest_simout.get('sldemo_absbrake_output');

% Plot wheel speed and car speed
subplot(3,1,1);
plot(out.get('yout').Values.Vs.Time, ...
    out.get('yout').Values.Vs.Data);
grid on;
title('Vehicle speed'); ylabel('Speed(rad/sec)'); xlabel('Time(sec)');
subplot(3,1,2);
plot(out.get('yout').Values.Ww.Time, ...
    out.get('yout').Values.Ww.Data);
grid on;
title('Wheel speed'); ylabel('Speed(rad/sec)'); xlabel('Time(sec)');
subplot(3,1,3);
plot(out.get('slp').Values.Time, ...
    out.get('slp').Values.Data);
grid on;
title('Slip'); xlabel('Time(sec)'); ylabel('Normalized Relative Slip');
```
5. Set the figure options for the test file `custom_figures`. Under **Test File Options**:

   - Select **Close all open figures at the end of execution**. This option closes figures created by your Test Manager MATLAB code.
   - Select **Store MATLAB figures**.
6. With the test case or the test file selected, click **Run**.
7. In the **Results and Artifacts** pane, select the test case under the results for this test run. Click the links under **MATLAB Figures** to see the plots generated when the test ran. The plot generated by the code you entered appears under **Custom Criteria**.



Include Figures in a Report

You can select the **MATLAB Figures** option in the Create Test Results Report dialog box to include custom figures in your report. Alternatively, you can set report options under **Test File Options**. The **Test File Options** settings are saved with the test file.

1. Select the test file `custom_figures`.
2. Under **Test File Options**, select **Generate report after execution**. The section expands, displaying the same report options you can set using the dialog box.
3. To see the figures regardless of how the tests performed, set **Results for** to `All Tests`.
4. Select the **MATLAB figures** check box.
5. With the test file selected, run the test. Running the test generates the report and opens it in the PDF viewer.
6. Examine the report. The plot generated by the code you entered under **Custom Criteria** appears in the report section **Custom Criteria Plots**.

See Also

`sltest.testmanager.Options` | `sltest.testmanager.TestCase.getOptions` | `sltest.testmanager.TestFile.getOptions` | `sltest.testmanager.TestSuite.getOptions`

Related Examples

- “Export Test Results and Generate Reports” on page 8-9

Test Models Using MATLAB Unit Test

In this section...

You can use the MATLAB Unit Test framework to run tests authored in Simulink Test. Using the MATLAB Unit Test framework:

- Allows you to execute model tests together with MATLAB Unit Test scripts, functions, and classes.
- Enables model and code testing using the same framework.
- Enables integration with continuous integration (CI) systems, such as Jenkins™.

Overall Workflow

To run tests with MATLAB Unit Test:

1. Create a `TestSuite` from the Simulink Test file.
2. Create a `TestRunner`.
3. Create plugin objects to customize the `TestRunner`. For example:
    - The `matlab.unittest.plugins.TAPPlugin` produces a results stream according to the Test Anything Protocol for use with certain CI systems.
    - The `sltest.plugins.ModelCoveragePlugin` specifies model coverage collection and makes coverage results accessible from the command line.
4. Add the plugins to the `TestRunner`.
5. Run the test using the `run` method, or run tests in parallel using the `runInParallel` method.

Considerations

When running tests using MATLAB Unit Test, consider the following:

- If you disable a test in the Test Manager, the test is filtered using MATLAB Unit Test, and the result reflects a failed assumption.

Comparison of Test Nomenclature

MATLAB Unit Test has analogous properties to the functionality in Simulink Test. For example,

- If the test case contains iterations, the MATLAB Unit Test contains parameterizations.
- If the test file or test suite contains callbacks, the MATLAB Unit Test contains one or more callbacks fixtures.

Test Case Iterations and MATLAB Unit Test parameterizations

parameterization details correspond to properties of the iteration.

Simulink Test	MATLAB Unit Test
Iteration type: Scripted	parameterization property: `ScriptedIteration`
Iteration type: Table	parameterization property: `TableIteration`
Iteration name	parameterization Name
Test case iteration object	parameterization Value

Test Callbacks and MATLAB Unit Test Fixtures

Fixtures depend on callbacks contained in the test file. Fixtures do not include test case callbacks, which are executed with the test case itself.

Callbacks in Simulink Test	Fixtures in MATLAB Unit Test
Test file callbacks	`FileCallbacksFixture`
Test suite callbacks	`SuiteCallbacksFixture`
File and suite callbacks	Heterogeneous `CallbacksFixture`, containing `FileCallbacksFixture` and `SuiteCallbacksFixture`
No callbacks	No fixture

Basic Workflow Using MATLAB® Unit Test

This example shows how to create and run a basic MATLAB® Unit Test for a test file created in Simulink® Test™. You create a test suite, run the test, and display the diagnostic report.

Before running this example, temporarily disable warnings that result from verification failures.

```
warning off Stateflow:Runtime:TestVerificationFailed;
warning off Stateflow:cdr:VerifyDangerousComparison;
```

1. Author a test file in the Test Manager, or start with a preexisting test file. For this example, `AutopilotTestFile` tests a component of an autopilot system against several requirements, using `verify` statements.

2. Create a `TestSuite` from the test file.

```
apsuite = testsuite('AutopilotTestFile.mldatx');
```

3. Run the test, creating a `TestResult` object. The command window returns warnings from the `verify` statement failures.

```
apresults = run(apsuite);
```

```
Setting up FileCallbacksFixture
Done setting up FileCallbacksFixture: Invoked setup callback of "AutopilotTestFile".
__________

Running AutopilotTestFile > Basic Design Test Cases

================================================================================
Verification failed in AutopilotTestFile > Basic Design Test Cases/Requirement 1.3 Tes
    ---------------------
    Framework Diagnostic:
    ---------------------
    Failed criteria: Verification
    --> Simulink Test Manager Results:
            Results: 2019-Mar-03 16:51:29/AutopilotTestFile/Basic Design Test Cases/Re
================================================================================
.
Done AutopilotTestFile > Basic Design Test Cases
__________
```

```
Tearing down FileCallbacksFixture
Done tearing down FileCallbacksFixture: Invoked cleanup callback of "AutopilotTestFile"
__________

Failure Summary:

     Name                                                              Failed  Incomple
    ===================================================================================
     AutopilotTestFile > Basic Design Test Cases/Requirement 1.3 Test    X
```

4. To view the details of the test, display the `Report` property of the `DiagnosticRecord` object. The record shows that a verification failed during the test.

```
apresults.Details.DiagnosticRecord.Report
```

```
ans =
    'Verification failed in AutopilotTestFile > Basic Design Test Cases/Requirement 1.3
         ---------------------
         Framework Diagnostic:
         ---------------------
         Failed criteria: Verification
         --> Simulink Test Manager Results:
                 Results: 2019-Mar-03 16:51:29/AutopilotTestFile/Basic Design Test Case
```

Enable warnings.

```
warning on Stateflow:Runtime:TestVerificationFailed;
warning on Stateflow:cdr:VerifyDangerousComparison;
```

See Also

`Test` | `TestResult` | `TestRunner` | `TestSuite` | `matlab.unittest.plugins Package`

Related Examples

- "Output Results for Continuous Integration Systems" on page 7-129
- "Run Tests for Various Workflows" (MATLAB)

Output Results for Continuous Integration Systems

In this section...

You can create model tests that are compatible with continuous integration (CI) systems such as Jenkins. To create CI-compatible results, run your Simulink Test files using MATLAB Unit Test.

To run CI-compatible tests, follow this general procedure:

1. Create a test suite from the MLDATX test file.
2. Create a test runner.
3. Create plugins for the test output or coverage results.

   - For test outputs, use the `TAPPlugin` or `XMLPlugin`.
   - For model coverage, use the `ModelCoveragePlugin` and `CoberturaFormat`. When collecting model coverage in Cobertura format:

     - Only top model coverage is reflected in the Cobertura XML.
     - Only model Decision coverage is reflected, and it is mapped to Condition elements in Cobertura XML.
4. Create plugins for CI-compatible output.
5. Add the plugins to the test output or coverage results.
6. Add the test output plugins or coverage result plugins to the test runner.
7. Run the test.

Test a Model for Continuous Integration Systems

This example shows how to test a model, publish Test Manager results, and output results in TAP format with a single execution.

You use MATLAB® Unit Test to create a test suite and a test runner, and customize the runner with these plugins:

- `matlab.unittest.plugins.TestReportPlugin` produces a MATLAB Test Report.
- `sltest.plugins.TestManagerResultsPlugin` adds Test Manager results to the MATLAB Test Report.

- `matlab.unittest.plugins.TAPPlugin` outputs results to a TAP file.

The test case creates a square wave input to a controller subsystem and sweeps through 25 iterations of parameters `a` and `b`. The test compares the `alpha` output to a baseline with a tolerance of `0.0046`. The test fails on those iterations in which the output exceeds this tolerance.

Before running this example, ensure that the working folder is writable.

1. Open the Simulink® Test™ test file.

```
testfile = fullfile('f14ParameterSweepTest.mldatx');
sltest.testmanager.view;
sltest.testmanager.load(testfile);
```

2. In the Test Manager, configure the test file for reporting.

Under **Test File Options**, select **Generate report after execution**. The section expands, displaying several report options. For more information, see “Save Reporting Options with a Test File” on page 8-10.

3. Create a test suite from the Simulink® Test™ test file.

```
import matlab.unittest.TestSuite

suite = testsuite('f14ParameterSweepTest.mldatx');
```

4. Create a test runner.

```
import matlab.unittest.TestRunner

f14runner = TestRunner.withNoPlugins;
```

5. Add the `TestReportPlugin` to the test runner.

The plugin produces a MATLAB Test Report `F14Report.pdf`.

```
import matlab.unittest.plugins.TestReportPlugin

pdfFile = 'F14Report.pdf';
trp = TestReportPlugin.producingPDF(pdfFile);
addPlugin(f14runner,trp)
```

6. Add the `TestManagerResultsPlugin` to the test runner.

The plugin adds Test Manager results to the MATLAB Test Report.

```
import sltest.plugins.TestManagerResultsPlugin

tmr = TestManagerResultsPlugin;
addPlugin(f14runner,tmr)
```

7. Add the `TAPPlugin` to the test runner.

The plugin outputs to the `F14Output.tap` file.

```
import matlab.unittest.plugins.TAPPlugin
import matlab.unittest.plugins.ToFile

tapFile = 'F14Output.tap';
tap = TAPPlugin.producingVersion13(ToFile(tapFile));
addPlugin(f14runner,tap)
```

8. Run the test.

Several iterations fail, in which the signal-baseline difference exceeds the tolerance criteria.

```
result = run(f14runner,suite);
```

```
Generating test report. Please wait.
    Preparing content for the test report.

    Adding content to the test report.
    Writing test report to file.
Test report has been saved to:
 C:\TEMP\Bdoc19a_1067994_10284\ibE7461B\27\tpc8279f68\simulinktest-ex40056435\F14Repor
```

A single execution of the test runner produces two reports:

- A MATLAB Test Report that contains Test Manager results.
- A TAP format file that you can use with CI systems.

```
sltest.testmanager.clearResults
sltest.testmanager.clear
sltest.testmanager.close
```

Model Coverage Results for Continuous Integration

This example shows how to generate model coverage results for use with continuous integration. Coverage is reported in the Cobertura format. You run a Simulink® Test™ test file using MATLAB® Unit Test.

1. Import classes and create a test suite from the test file `AutopilotTestFile.mldatx`.

```
import matlab.unittest.TestRunner

aptest = sltest.testmanager.TestFile(fullfile(matlabroot,'toolbox','simulinktest',...
    'simulinktestdemos','AutopilotTestFile.mldatx'));
apsuite = testsuite(aptest.FilePath);
```

2. Create a test runner.

```
trun = TestRunner.withNoPlugins;
```

3. Set the coverage metrics to collect. This example uses decision coverage. In the Cobertura output, decision coverage is listed as condition elements.

```
import sltest.plugins.coverage.CoverageMetrics

cmet = CoverageMetrics('Decision',true);
```

4. Set the coverage report properties. This example produces a file `R13Coverage.xml` in the current working folder. Ensure your working folder has write permissions.

```
import sltest.plugins.coverage.ModelCoverageReport
import matlab.unittest.plugins.codecoverage.CoberturaFormat

rptfile = 'R13Coverage.xml';
rpt = CoberturaFormat(rptfile)
```

```
rpt = 
  CoberturaFormat with no properties.
```

5. Create a model coverage plugin. The plugin collects the coverage metrics and produces the Cobertura format report.

```
import sltest.plugins.ModelCoveragePlugin

mcp = ModelCoveragePlugin('Collecting',cmet,'Producing',rpt)
mcp = 
  ModelCoveragePlugin with properties:

    RecordModelReferenceCoverage: '<default>'
                 MetricsSettings: [1x1 sltest.plugins.coverage.CoverageMetrics]
```

6. Add the coverage plugin to the test runner.

```
addPlugin(trun,mcp)

% Turn off command line warnings:
warning off Stateflow:cdr:VerifyDangerousComparison
warning off Stateflow:Runtime:TestVerificationFailed
```

7. Run the test.

```
APResult = run(trun,apsuite)
```

```
APResult = 
  TestResult with properties:

          Name: 'AutopilotTestFile > Basic Design Test Cases/Requirement 1.3 Test'
        Passed: 0
        Failed: 1
    Incomplete: 0
      Duration: 7.7443
       Details: [1x1 struct]

Totals:
   0 Passed, 1 Failed, 0 Incomplete.
   7.7443 seconds testing time.
```

8. Reenable warnings.

```
warning on Stateflow:cdr:VerifyDangerousComparison
warning on Stateflow:Runtime:TestVerificationFailed
```

See Also

TestRunner | TestSuite | matlab.unittest.plugins.TAPPlugin | matlab.unittest.plugins.TestReportPlugin |

`sltest.plugins.ModelCoveragePlugin` | `sltest.plugins.TestManagerResultsPlugin`

More About

- “Test Models Using MATLAB Unit Test” on page 7-125

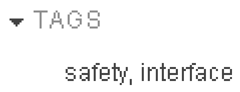
Filter Test Execution and Results

In this section...
"Add Tags" on page 7-135
"Filter Tests and Results" on page 7-135
"Run Filtered Tests" on page 7-135

You can run a subset of tests or view a subset of test results by filtering test tags. Tags are a property of the test case, test suite, or test file.

Add Tags

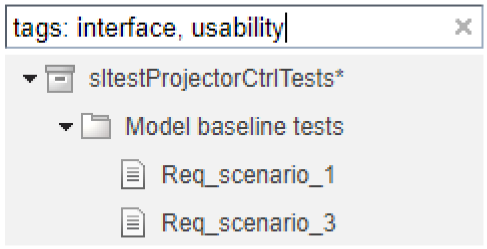
Add comma-separated tags to the **Tags** section in the Test Browser. Tags cannot contain spaces; spaces are corrected to commas.



safety, interface

Filter Tests and Results

In the text box at the top of the **Test Browser** or **Results and Artifacts** pane, filter tests by entering `tags: id1, id2, ...` where `id1` and `id2` are example test tags. Enter multiple tags separated by commas to return tests containing any tag in the list.



tags: interface, usability

Run Filtered Tests

To run a subset of tests

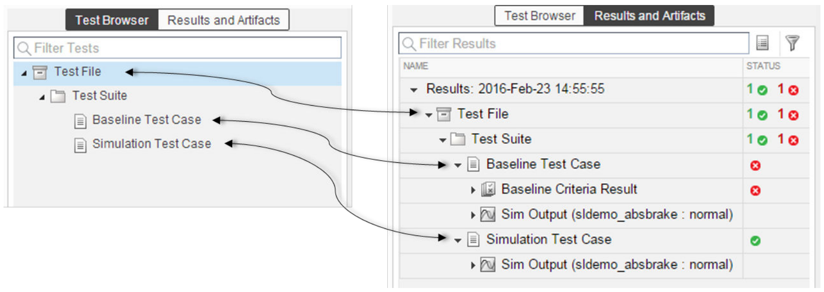
1. Filter the tests using tags.
2. In the toolstrip, click the down arrow below **Run** and select **Run Filtered**.

Test Manager Results and Reports

View Test Case Results

In this section...
"View Results Summary" on page 8-2
"Visualize Test Case Simulation Output and Criteria" on page 8-4

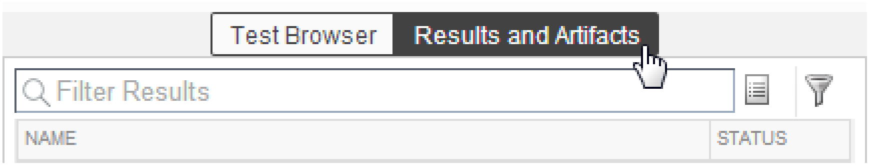
After a test case has finished running in the Test Manager, the test case result becomes available in the **Results and Artifacts** pane. Test results are organized in the same hierarchy as the test file, test suite, and test cases that were run from the **Test Browser** pane. In addition, the **Results and Artifacts** pane shows the criteria results and simulation output, if applicable to the test case.



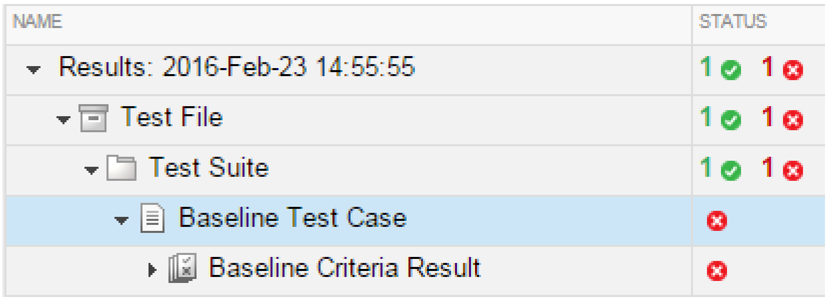
View Results Summary

The test case results tab gives a high-level summary and other information about an individual test case result. To open the test case results tab:

1. Select the **Results and Artifacts** pane.

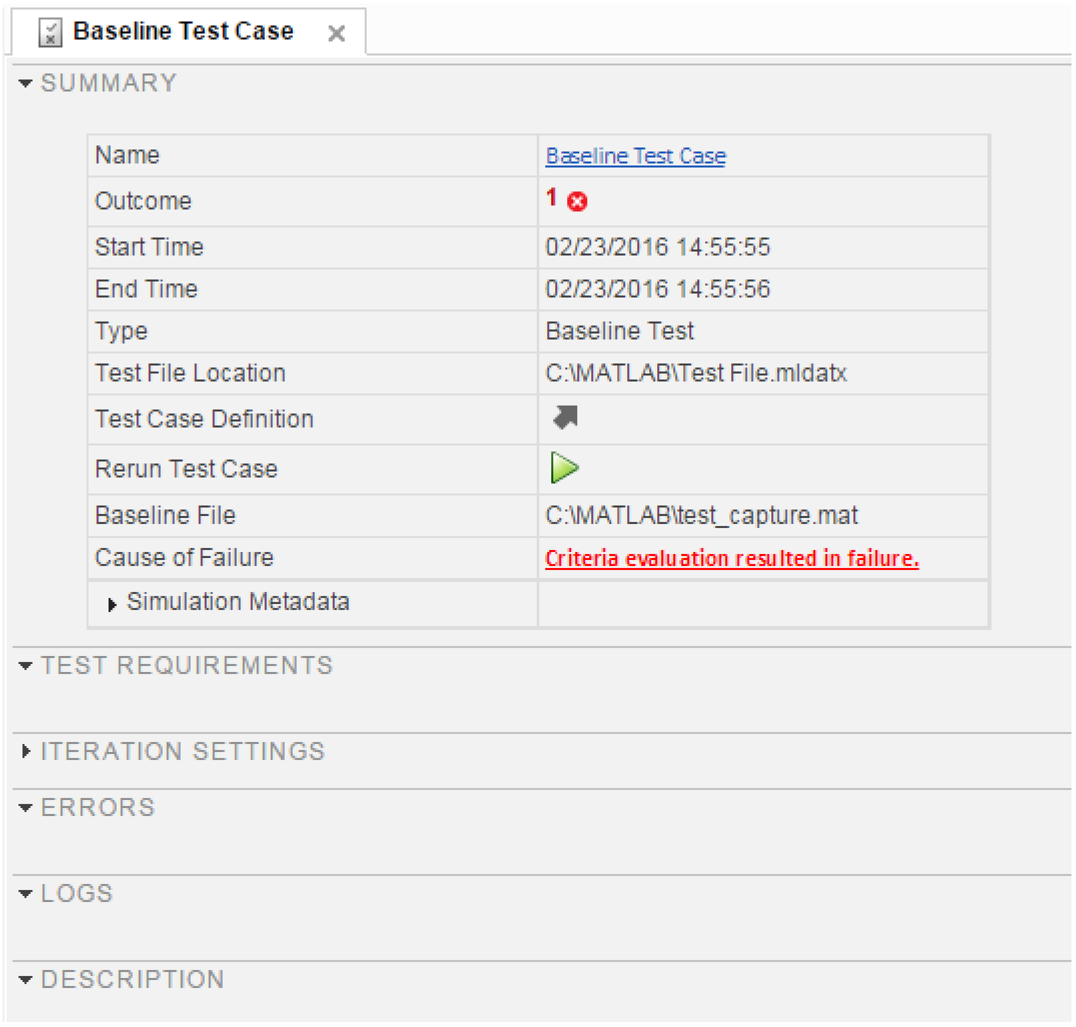


2. Double-click a test case result.



NAME	STATUS
Results: 2016-Feb-23 14:55:55	1 ✓ 1 ✗
Test File	1 ✓ 1 ✗
Test Suite	1 ✓ 1 ✗
Baseline Test Case	✗
Baseline Criteria Result	✗

A tab opens containing the test case results information.



Name	Baseline Test Case
Outcome	1 ⊗
Start Time	02/23/2016 14:55:55
End Time	02/23/2016 14:55:56
Type	Baseline Test
Test File Location	C:\MATLAB\Test File.mldatx
Test Case Definition	
Rerun Test Case	▷
Baseline File	C:\MATLAB\test_capture.mat
Cause of Failure	Criteria evaluation resulted in failure.
Simulation Metadata	

Visualize Test Case Simulation Output and Criteria

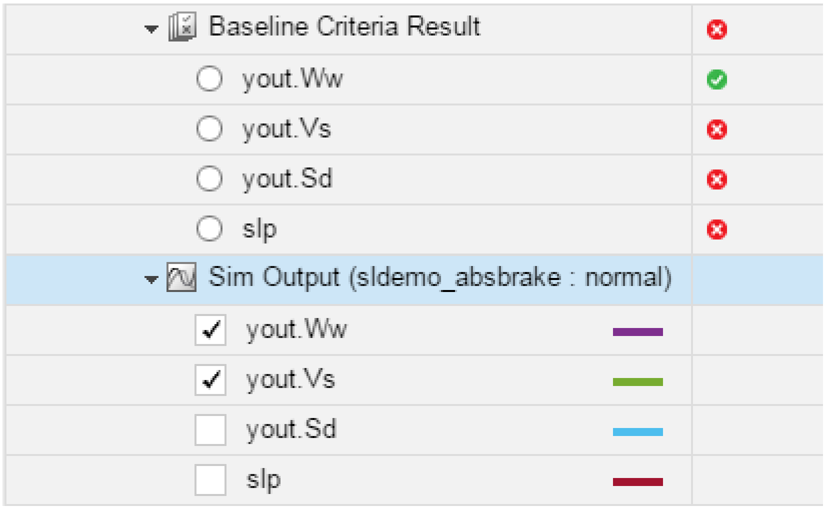
You can view signal data from simulation output or comparisons of signal data used in baseline or equivalence criteria.

To view simulation output from a test case:

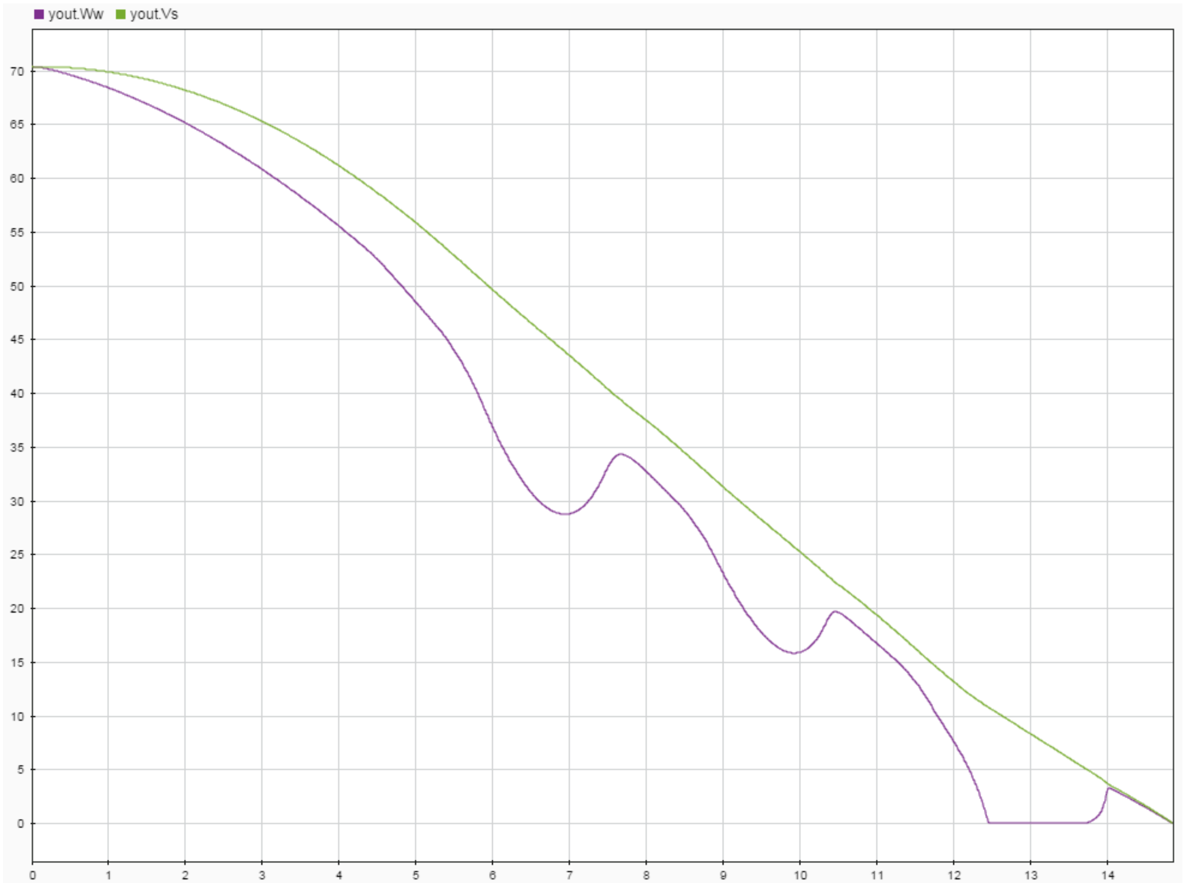
1 Select the **Results and Artifacts** pane.

2 Expand the **Sim Output** section of the test case result.

3 Select the check box of signals you want to plot.

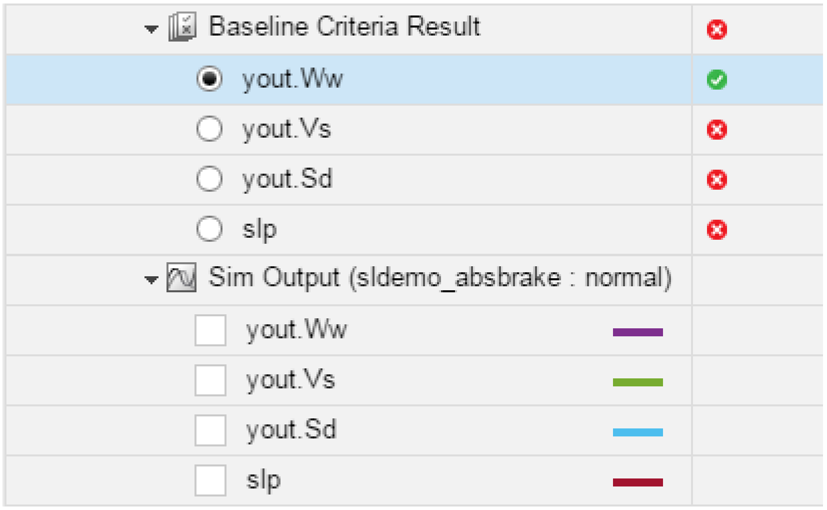


The **Visualize** tab appears and plots the signals.



To view equivalence or baseline criteria comparisons:

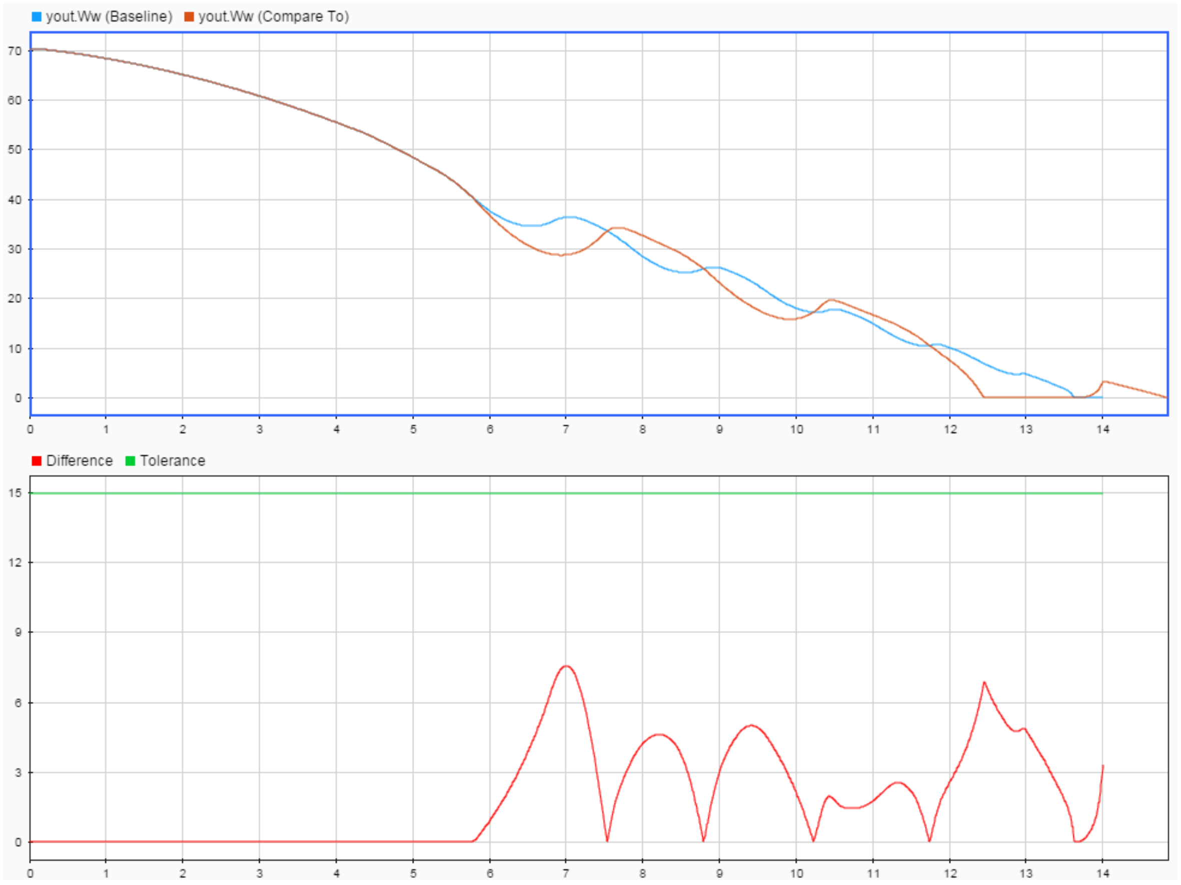
1. Select the **Results and Artifacts** pane.
2. Expand the **Baseline Criteria Result** or **Equivalence Criteria Result** section of the test case result.
3. Select the option button of the signal comparison you want to plot.



Baseline Criteria Result ✗
- ● yout.Ww ✓
- ○ yout.Vs ✗
- ○ yout.Sd ✗
- ○ slp ✗

Sim Output (sldemo_absbrake : normal)
- ☐ yout.Ww
- ☐ yout.Vs
- ☐ yout.Sd
- ☐ slp

The **Comparison** tab appears and plots the signal comparison.



To see an example of creating a test case and viewing the results, see “Compare Model Output To Baseline Data” on page 7-9.

Note When you run a test multiple times, by default the new signals are added to the plot from previous test runs. To instead overwrite the plots with only the new results, right-click **Sim Output** and select **Plot Signals > Overwrite**.

Export Test Results and Generate Reports

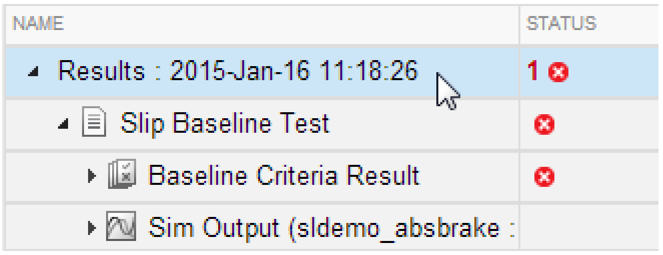
In this section...

Once you have run test cases and generated test results, you can export results and generate reports. Test case results appear in the **Results and Artifacts** pane.

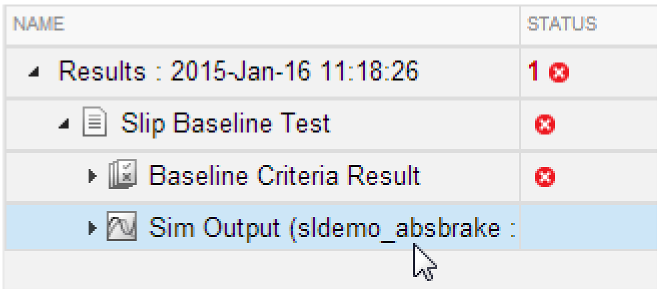
Export Results

Test results are saved separately from the test file. To save results, select the result in the **Results and Artifacts** pane, and click **Export** on the toolstrip.

- Select complete result sets to export to a MATLAB data export file (`.mldatx`).



- Select criteria comparisons or simulation output to export signal data to the base workspace or to a MAT-file.



NAME	STATUS
Results : 2015-Jan-16 11:18:26	1 ✗
Slip Baseline Test	✗
Baseline Criteria Result	✗
Sim Output (sldemo_absbrake :	

Create a Test Results Report

Result reports contain report overview information, the test environment, results summaries with test outcomes, comparison criteria plots, and simulation output plots. You can customize the information included in the report, and you can save the report in three different file formats: `ZIP (HTML)`, `DOCX`, and `PDF`.

1 In the **Results and Artifacts** pane, select results for a test file, test suite, or test case.

   Note You can create a report from multiple result sets, but you cannot create a report from multiple test files, test suites, or test cases within results sets.

2 From the toolstrip, click **Report**.
3 Select the options to specify report contents.
4 Set **File Format** to the output format you want.
5 Click **Create**.

Save Reporting Options with a Test File

You can generate a report every time you run a test case in a test file, using the same report settings each time. To generate a report each time you run the test, set options under **Test File Options**. These settings are saved with the test file.

1 In the **Test Browser** pane, select the test file whose report options you want to set.
2 Under **Test File Options**, select **Generate report after execution**. The section expands, displaying the same report options you can set using the dialog box.

3. Set the options. To include figures generated by callbacks or custom criteria, select **MATLAB figures**. For more information, see “Create, Store, and Open MATLAB Figures” on page 7-122.
4. Store the settings with your test file. Save the test file.
5. If you want to generate a report using these settings, select the test file and run the test.

Generate Reports Using Templates

Microsoft Word Format

If you have a MATLAB Report Generator license, you can create reports from a Microsoft Word template. The report can be a Microsoft Word or PDF document.

The report generator in Simulink Test fills information into rich text content controls in your Microsoft Word template document. For more information on how to use rich text content controls or customize part templates, see the MATLAB Report Generator documentation, such as “Add Holes in a Microsoft Word Template” (MATLAB Report Generator).

For a sample template, go to the path:

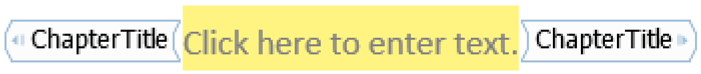
```
cd(matlabroot);
cd('help\toolbox\sltest\examples');
```

In the `examples` folder, open the file `Template.dotx`.

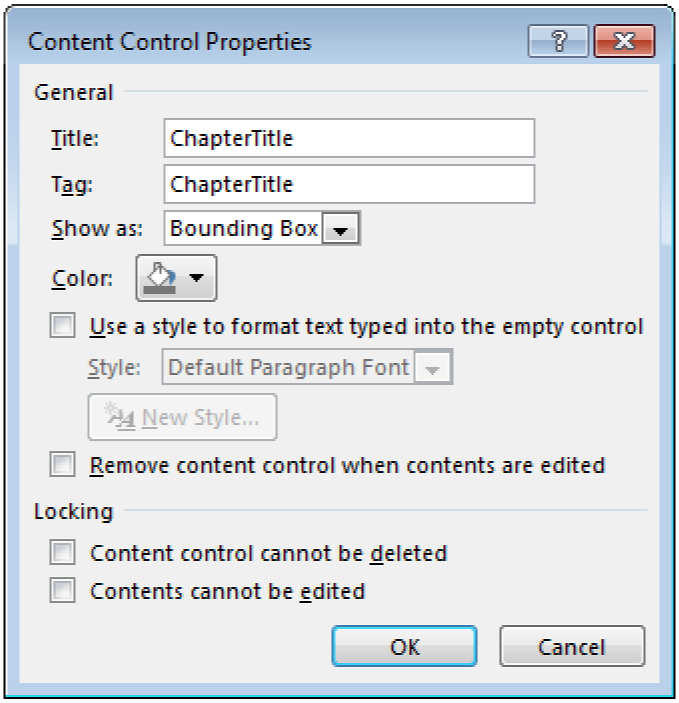
In the Microsoft Word template, you can add rich text content controls. Each Simulink Test report section can be inserted into the rich text content controls. The control names are:

- `ChapterTitle` — report title
- `ChapterTestPlatform` — version of MATLAB used to execute tests
- `ChapterTOC` — test results table of contents
- `ChapterBody` — test results

For example, the chapter title rich text content control appears in the Microsoft Word template as:



To change the control name, right-click the rich text content control and select **Properties**. Specify the control name, `ChapterTitle` or other name, in the **Title** and **Tag** field.


Content Control Properties

General

Title: ChapterTitle

Tag: ChapterTitle

Show as: Bounding Box

Color:

[ ] Use a style to format text typed into the empty control

Style: Default Paragraph Font

[ ] Remove content control when contents are edited

Locking

[ ] Content control cannot be deleted

[ ] Contents cannot be edited

To generate a report from the Test Manager using a Microsoft Word template:

1. In the Test Manager, select the **Results and Artifacts** pane.
2. Select results for a test file, test suite, or test case in the **Results and Artifacts** pane.
3. From the toolstrip, click **Report**.
4. Select the report options.
5. Select `DOCX` or `PDF` for the **File Format**.
6. Specify the full path and file name of your Microsoft Word template.
7. Click **Create**.

PDF or HTML Formats

If you have a MATLAB Report Generator license, you can create reports from a PDF or HTML template, using a PDFTX or HTMTX file. To generate a report from the Test Manager using a PDF or HTML template:

1. In the Test Manager, select the **Results and Artifacts** pane.
2. Select results for a test file, test suite, or test case in the **Results and Artifacts** pane.
3. From the toolstrip, click **Report**.
4. Select the report options.
5. Select ZIP or PDF for the **File Format**. Selecting ZIP generates an HTML report.
6. Specify the full path and file name of your template. For PDF, use a PDFTX file. For HTML, use an HTMTX file. For more information on creating templates, see "Templates" (MATLAB Report Generator).
7. Click **Create**.

Generating a Test Results Report

Report test results for a baseline test.

This example shows how to generate a test results report from the Test Manager using a baseline test case. The model used for this example is sltestTestManagerReportsExample. Switch to a directory with write permissions.

Load and run the test file

Load and run the test file programmatically using the Test Manager. The test file contains a baseline test case that fails when it is run. The baseline criteria specified in the baseline test case does not match the model simulation, which makes the test case fail.

```
exampleFile = fullfile(matlabroot, ...
                     'toolbox', 'simulinktest', 'simulinktestdemos', ...
                     'sltestTestManagerReportsTestSuite.mldatx');
sltest.testmanager.load(exampleFile);
baselineObj = sltest.testmanager.run;
```

Generate the report

Generate a report of the test case results using the results set object. The report is saved as a ZIP and will show all test results. The report opens when it is completed.

```
sltest.testmanager.report(baselineObj,'baselineReport.zip',...
    'IncludeTestResults',0, 'IncludeComparisonSignalPlots', true);
```

View the report when it is finished generating. Notice that the overall baseline test case fails. The signals in baseline criteria do not match, which causes the test failure. You can view the signal comparison plots in the report to verify the failure.

```
sltest.testmanager.clear;
sltest.testmanager.clearResults;
```

See Also

Related Examples

- "Templates" (MATLAB Report Generator)
- "Create, Store, and Open MATLAB Figures" on page 7-122

Customize Test Reports

In this section...

You can choose how to format and aggregate test results by customizing reports. Use the `sltest.testmanager.TestResultReport` class to create a subclass and then use the properties and methods to customize how the Test Manager generates the results report. You can change font styles, add plots, organize results into tables, include model images, and more. Using the custom class, requires a MATLAB Report Generator license.

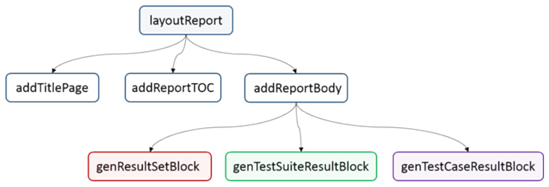
Inherit the Report Class

To customize the generated report, you must inherit from the `sltest.testmanager.TestResultReport` class. After you inherit from the class, you can modify the properties and methods. To inherit the class, add the class definition section to a new or existing MATLAB script. The subclass is your custom class name, and the superclass that you inherit from is `sltest.testmanager.TestResultReport`. For more information about creating subclasses, see "Design Subclass Constructors" (MATLAB). Then, add code to the inherited class or methods to create your customizations.

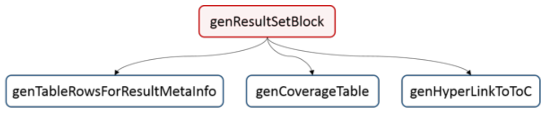
```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    %
    % Report customization code here
    %
end
```

Method Hierarchy

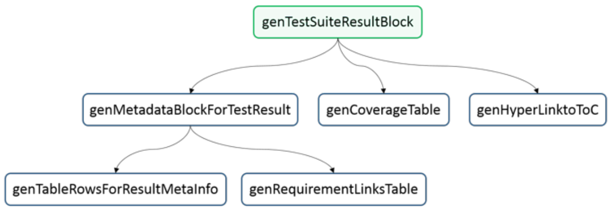
When you create the subclass, the derived class inherits methods from the `sltest.testmanager.TestResultReport` class. The body of the report is separated into three main groups: the result set block, the test suite result block, and the test case result block.



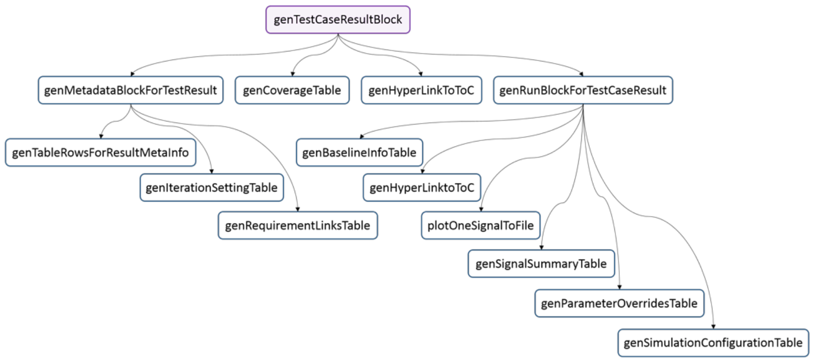
The result set block contains the result set table, the coverage table, and links to the table of contents.



The test suite result block contains the test suite results table, the coverage table, requirements links, and links to the table of contents.



The test case result block contains the test case and test iterations results table, the coverage table, requirements links, signal output plots, comparison plots, test case settings, and links to the table of contents.



Modify the Class

To insert your own report content or change the layout of the generated report, modify the inherited class methods. For general information about modifying methods, see “Modify Inherited Methods” (MATLAB).

A simple modification to the generated report could be to add some text to the title page. The method used here is `addTitlePage`.

```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    methods
        function this = CustomReport(resultObjects, reportFilePath)
            this@sltest.testmanager.TestResultReport(resultObjects,...
            reportFilePath);
        end
    end

    methods(Access=protected)
        function addTitlePage(obj)
            import mlreportgen.dom.*;

            % Add a custom message
            label = Text('Some custom content can be added here');
            append(obj.TitlePart,label);

            % Call the superclass method to get the default behavior
            addTitlePage@sltest.testmanager.TestResultReport(obj);
        end
    end
end
```

Click here for a code file of this example.

A more complex modification of the generated report is to include a snapshot of the model that was tested.

```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    methods
        function this = CustomReport(resultObjects,reportFilePath)
            this@sltest.testmanager.TestResultReport(resultObjects,reportFilePath);
        end
    end

    methods(Access=protected)
        % Method to customize test case/iteration result section in the report
        function docPart = genTestCaseResultBlock(obj,result)
            % result: A structure containing test case or iteration result
            import mlreportgen.dom.*;

            % Call the superclass method to get the default behavior
            docPart = genTestCaseResultBlock@sltest.testmanager.TestResultReport(...
                                                                 obj,result);

            % Get the test case result data for putting in the report
            tcrObj = result.Data;

            % Insert model screenshot at the test case result level
            if isa(tcrObj, 'sltest.testmanager.TestCaseResult')

                % Initialize model name
                modelName = '';

                % Check in the test case result if it has model information. If
                % not, it means there were iterations in the test case or a
                % model was not used.
                testSimMetaData = tcrObj.SimulationMetaData;

                if (~isempty(testSimMetaData))
                    modelName = testSimMetaData.modelName;
                end

                % Get iteration results
                iterResults = getIterationResults(tcrObj);

                % Get the model name in case test case had iterations
                if (~isempty(iterResults))
                    modelName = iterResults(1).SimulationMetaData.modelName;
                end

                % Insert model snapshot. This will not work for harnesses. With
                % minimal changes we can also open the harness used for
                % testing.
                if (~isempty(modelName))
                    try
                        open_system(modelName);
                        outputFileName = [tempdir, modelName, '.png'];
                        if exist(outputFileName,'file')
                            delete(outputFileName);
                        end
                        print(outputFileName, '-s', '-dpng');
                        para = sltest.testmanager.ReportUtility.genImageParagraph(...
                            outputFileName,...
                            '5.2in','3.7in');
                        append(docPart,para);
                    catch
                    end
                end
            end
        end
    end
end
```

Click here for a code file of this example.

Generate a Report Using the Custom Class

After you customize the class and methods, use the `sltest.testmanager.report` to generate the report. You must use the `'CustomReportClass'` name-value pair for the custom class, specified as a string. For example:

```
% Generate the result set from imported data
result = sltest.testmanager.importResults('demoResults.mldatx');

% Specify the report file name and path
filePath = 'testreport.zip';

% Generate the report using the custom class
sltest.testmanager.report(result,filePath, ...
            'Author','MathWorks',...
            'Title','Test',...
            'IncludeMLVersion',true,...
            'IncludeTestResults',int32(0),...
            'CustomReportClass','CustomReport',...
            'LaunchReport', true);
```

Alternatively, you can create your custom report using the Test Manager report dialog box. Select a test result, click the **Report** button on the toolstrip, and specify the custom report class in the Create Test Result Report dialog box. For the Test Manager to use the custom report class, the class must be on the MATLAB path.

See Also

`sltest.testmanager.TestResultReport` | `sltest.testmanager.report`

Related Examples

- “Design Subclass Constructors” (MATLAB)

Append Code to a Test Report

This example shows how to use a customization class to print integrated code in a test results report. If you test models that include handwritten code, you can print the code to a report to be reviewed with the test results.

The cruise control model integrates handwritten C code using an S-Function builder block. The C code is a utility function that disregards simultaneous pressing of two buttons: `Accel/Res` and `Coast/Set`.

This example requires Simulink® Report Generator™ and Microsoft® Windows.

Example Files

Before running this example, add the example folders to the path and set the filenames.

```
addpath(fullfile(matlabroot,'examples','simulinktest'));
addpath(fullfile(matlabroot,'examples','simulinktest','main'));
rptCustom = 'textAppendReport.m';
resultsFile = 'DoublePressSfcnSimTestResults';
filePath = fullfile(tempdir,'textAppendedReport.zip');
```

Report Customization Class

The report customization class `textAppendReport.m` appends the S-Function code to the end of the report body.

```
open(rptCustom)
```

Load the Results and Create the Report

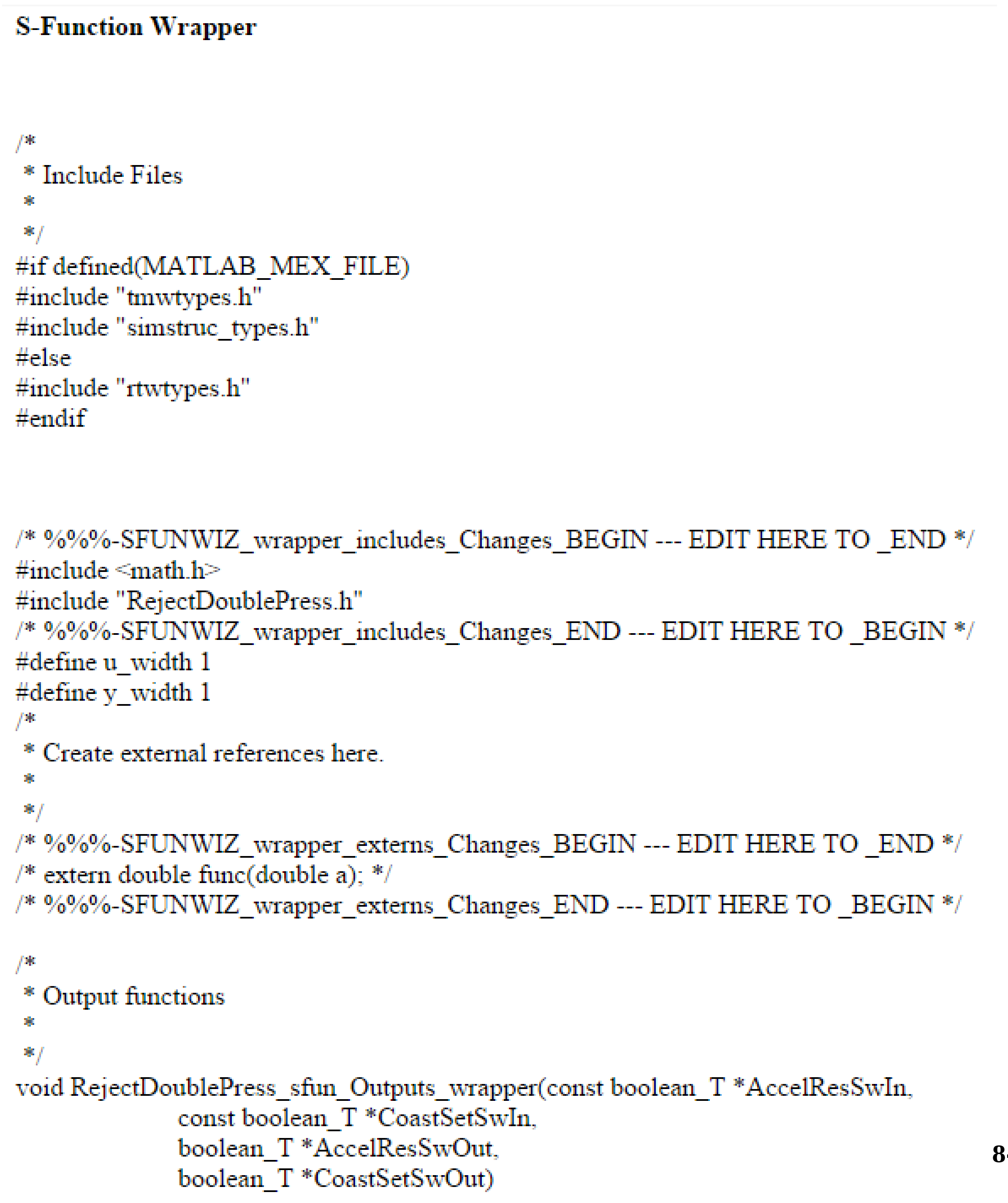
1. Load the test results file.

```
result = sltest.testmanager.importResults(resultsFile);
```

2. Create the test report using the customization.

```
sltest.testmanager.report(result,filePath,'CustomReportClass','textAppendReport',...
    'IncludeTestResults',0)
```

3. The report appends the S-Function wrapper code:



S-Function Wrapper

```
/*
 * Include Files
 *
 */
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif



/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
#include <math.h>
#include "RejectDoublePress.h"
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width 1
#define y_width 1
/*
 * Create external references here.
 *
 */
/* %%%-SFUNWIZ_wrapper_externs_Changes_BEGIN --- EDIT HERE TO _END */
/* extern double func(double a); */
/* %%%-SFUNWIZ_wrapper_externs_Changes_END --- EDIT HERE TO _BEGIN */

/*
 * Output functions
 *
 */
void RejectDoublePress_sfun_Outputs_wrapper(const boolean_T *AccelResSwIn,
                    const boolean_T *CoastSetSwIn,
                    boolean_T *AccelResSwOut,
                    boolean_T *CoastSetSwOut)
```

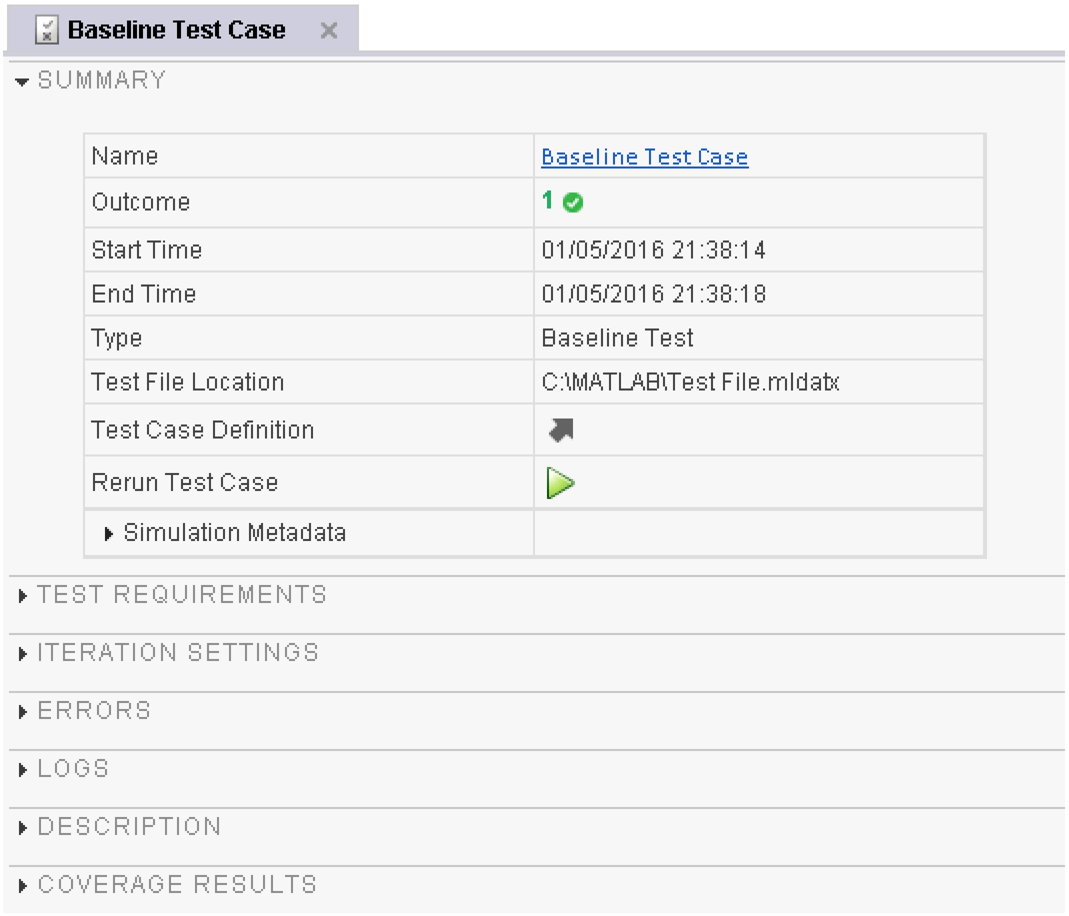
For more information on report customization, see “Customize Test Reports” on page 8-15.

```
rmpath(fullfile(matlabroot,'examples','simulinktest'));
rmpath(fullfile(matlabroot,'examples','simulinktest','main'));
sltest.testmanager.clearResults;
sltest.testmanager.close;
```

Results Sections

In this section...

Double-click a test case results in the **Results and Artifacts** pane to open a results tab and view the test case result sections. A baseline test case result is shown as an example.



Name	Baseline Test Case
Outcome	1 ✓
Start Time	01/05/2016 21:38:14
End Time	01/05/2016 21:38:18
Type	Baseline Test
Test File Location	C:\MATLAB\Test File.mldatx
Test Case Definition	
Rerun Test Case	▶
▸ Simulation Metadata	

Summary

The **Summary** section includes the basic test information and the test outcome. For more information about the simulation, toggle the **Simulation Metadata** arrow to expand the section.

Test Requirements

A list of test requirements linked to the test case. See “Requirements” on page 7-97 for more information on linking requirements to test cases.

Iteration Settings

If you are using iterations to run test cases, then this section appears in the results. For more information about test iterations, see “Test Iterations” on page 7-71.

Errors

This section displays simulation errors captured from the Simulink Diagnostic Viewer. Errors from incorrect information defined in the test case and callback scripts are also shown here.

Logs

This section displays simulation warnings captured from the Simulink Diagnostic Viewer.

Description

You can include notes about the test results here. These notes are saved with the results.

Parameter Overrides

A list of parameter overrides specified in the test case under **Parameter Overrides**. If parameter overrides are not specified, then this section is not shown in the results summary.

Coverage Results

If you collect coverage in your test, then the coverage results appear in this section. Coverage results are aggregated at the test file, test suite, and test file level. For more information about coverage, see “Collect Coverage in Tests” on page 7-83.

Real-Time Testing

Test Models in Real Time

In this section...

You can test your system in environments that resemble your application. You begin with model simulation on a development computer, then use software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. Real-time testing executes an application on a standalone target computer that can connect to a physical system. Real-time testing can include effects of timing, signal interfaces, system response, and production hardware.

Real-time testing includes:

- Rapid prototyping, which tests a system on a standalone target connected to plant hardware. You verify the real-time tests against requirements and model results. Using rapid prototyping results, you can change your model and update your requirements, after which you retest on the standalone target.
- Hardware-in-the-loop (HIL), which tests a system that has passed several stages of verification, typically SIL and PIL simulations.

Overall Workflow

This example workflow describes the major steps of creating and executing a real-time test:

1. Create test cases that verify the model against requirements. Run the model simulation tests and save the baseline data.
2. Set up the real-time target computer.
3. Create test harnesses for real-time testing, or reuse model simulation test harnesses. In Test Sequence or Test Assessment blocks, `verify` statements assess the real-time

execution. In the test harnesses, use target and host scopes to display signals during execution.

4. In the Test Manager, create real-time test cases.
5. For the real-time test cases, configure target settings, inputs, callbacks, and iterations. Add baseline or equivalence criteria.
6. Execute the real-time tests.
7. Analyze the results in the Test Manager. Report the results.

Real-Time Testing Considerations

- Baseline or equivalence comparisons can fail because of missing data or time-shifted data from the real-time target computer. When investigating real-time test failures, look for time shifts or missing data points.
- You cannot override the real-time execution sample time for applications built from models containing a Test Sequence block. The code generated for the Test Sequence block contains a hard-coded sample time. Overriding the target computer sample time can produce unexpected results.
- Your target computer must have a file system to use `verify` statements and test case logging.

Complete Basic Model Testing

Real-time testing often takes longer than comparative model testing, especially if you execute a suite of real-time tests that cover several scenarios. Before executing real-time tests, complete requirements-based testing using desktop simulation. Using the desktop simulation results:

- Debug your model or make design changes that meet requirements.
- Debug your test sequence. Use the debugging features in the Test Sequence Editor. See “Debug a Test Sequence” on page 3-69.
- Update your requirements and add corresponding test cases.

Set up the Target Computer

Real-time testing requires a standalone target computer. Simulink Test only supports target computers running Simulink Real-Time. For more information, see:

- “Development Computer Setup and Configuration” (Simulink Real-Time)
- “Troubleshooting in Simulink Real-Time” (Simulink Real-Time)

Configure the Model or Test Harness

Real-time applications require specific configuration parameters and signal properties.

Code Generation

A real-time test case requires a real-time system target file. In the model or harness configuration parameters, in the **Code Generation** pane, set the **System target file** to `slrt.tlc` to generate system target code.

If your model requires a different system target file, you can set the parameter using a test case or test suite callback. After the real-time test executes, set the parameter to its original setting with a cleanup callback. For example, this callback opens the model and sets the system target file parameter to `slrt.tlc` for the model `sltestProjectorController`.

```
open_system(fullfile(matlabroot,'toolbox','simulinktest',...
'simulinktestdemos','sltestProjectorController'));
set_param('sltestProjectorController','SystemTargetFile','slrt.tlc');
```

Data Import/Export Format

Models must use a data format other than `dataset`. To set the data format:

1. Open the configuration parameters.
2. Select the **Data Import/Export** pane.
3. Select the **Format**.

Log Signals from Real-Time Execution

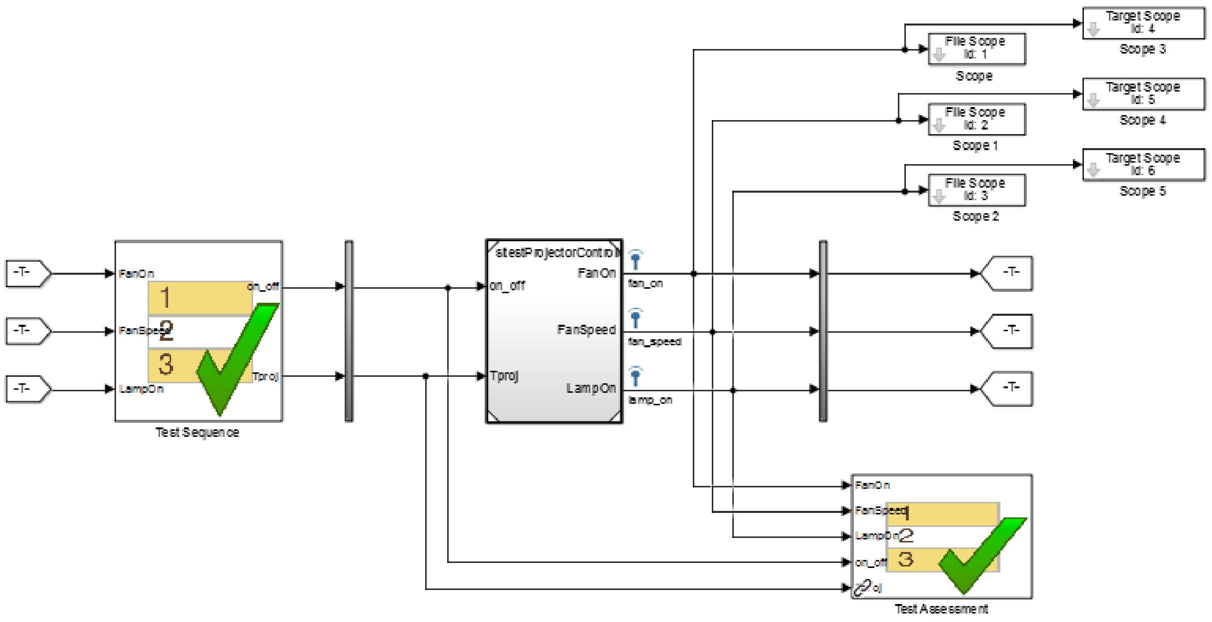
To configure your signals of interest for real-time testing:

- Enable signal logging in the Configuration Parameters, in the Data Import/Export pane.
- Connect signals to Scope blocks from the Simulink Real-Time block library. Set the **Scope type** property to `File`.

- Name each signal of interest using the signal properties. Unnamed signals can be assigned a default name which does not match the name of the baseline or equivalence signal.

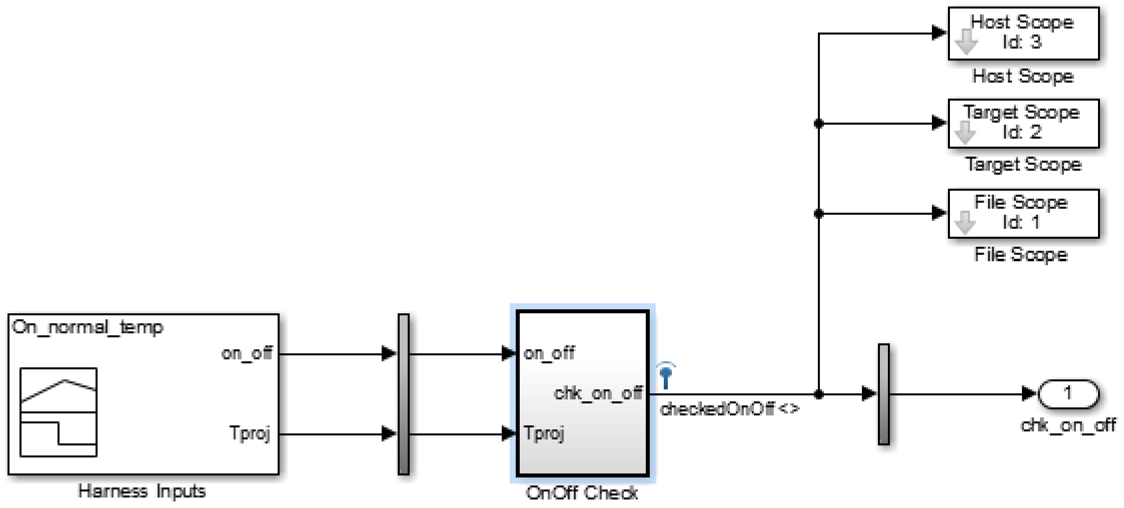
In this example test harness, the logged signals:

- Have explicit names.
- Use file scopes to return signal data to the Test Manager.
- Use target scopes to display data on the target computer.



View Signals During Real-Time Execution

To display signals on the target computer during real-time execution, add target scopes to your test harness. To display signals in the Simulink Real-Time Explorer, add host scopes. This test harness includes both target and host scopes for signal visualization. See Scope.



Add Test Cases for Real-Time Testing

Use the Test Manager to create real-time test cases. In the toolstrip, click **New > Real-Time Test**.

Test Type

You can select a baseline, equivalence, or simulation real-time test. For simulation test types, `verify` statements serve as pass/fail criteria in the test results. For equivalence and baseline test types, the equivalence or baseline criteria also serve as pass/fail criteria.

- **Baseline** — Compares the signal data returned from the target computer to the baseline in the test case. To compare a real-time execution result to a model simulation result, add the model baseline result to the real-time test case and apply optional tolerances to the signals.
- **Equivalence** — Compares signal data from a simulation and a real-time test, or two real-time tests. To run a real-time test on the target computer, then compare results to a model simulation:
  - Select **Simulation 1 on target**.

- Clear **Simulation 2 on target**.

  The test case displays two simulation sections, **Simulation 1** and **Simulation 2**.

  Comparing two real-time tests is similar, except that you select both simulations on target. In the **Equivalence Criteria** section, you can capture logged signals from the simulation and apply tolerances for pass/fail analysis.
- **Simulation**: Assesses the test result using only `verify` statements and real-time execution. If no `verify` statements fail, and the real-time test executes, the test case passes.

Load Application From

Using this option, specify how you want to load the real-time application. The real-time application is built from your model or test harness. You can load the application from:

- **Model** — Choose `Model` if you are running the real-time test for the first time, or your model changed since the last real-time execution. `Model` typically takes the longest because it includes model build and download. `Model` loads the application from the model, builds the real-time application, downloads it to the target computer, and executes it on the target computer.
- **Target Application** — Choose `Target Application` to send the target application from the host to a target computer, and execute the application. `Target Application` can be useful if you want to load an already-built application on multiple targets.
- **Target Computer** — This option executes an application that is already loaded on the real-time target computer. You can update the parameters in the test case and execute using `Target Computer`.

This table summarizes which steps and callbacks execute for each option.

Test Case Execution Step (first to last)	Load Application From		
	Model	Target Application	Target Computer
Executes pre-load callback	Yes	Yes	Yes
Loads Simulink model	Yes	No	No

Test Case Execution Step (first to last)	Load Application From		
	Model	Target Application	Target Computer
Executes post-load callback	Yes	No	No
Sets Signal Builder group	Yes	No	No
Builds real-time application from model	Yes	No	No
Downloads real-time application to target computer	Yes	Yes	No
Sets runtime parameters	Yes	Yes	Yes
Executes pre-start real-time callback	Yes	Yes	Yes
Executes real-time application	Yes	Yes	Yes
Executes cleanup callback	Yes	Yes	Yes

Model

Select the model from which to generate the real-time application.

Test Harness

If you use a test harness to generate the real-time application, select the test harness.

Simulation Settings Overrides

For real-time tests, you can override the simulation stop time, which can be useful in debugging a real-time test failure. Consider a 60-second test that returns a `verify` statement failure at 15 seconds due to a bug in the model. After debugging your model, you execute the real-time test to verify the fix. You can override the stop time to terminate the execution at 20 seconds, which reduces the time it takes to verify the fix.

Callbacks

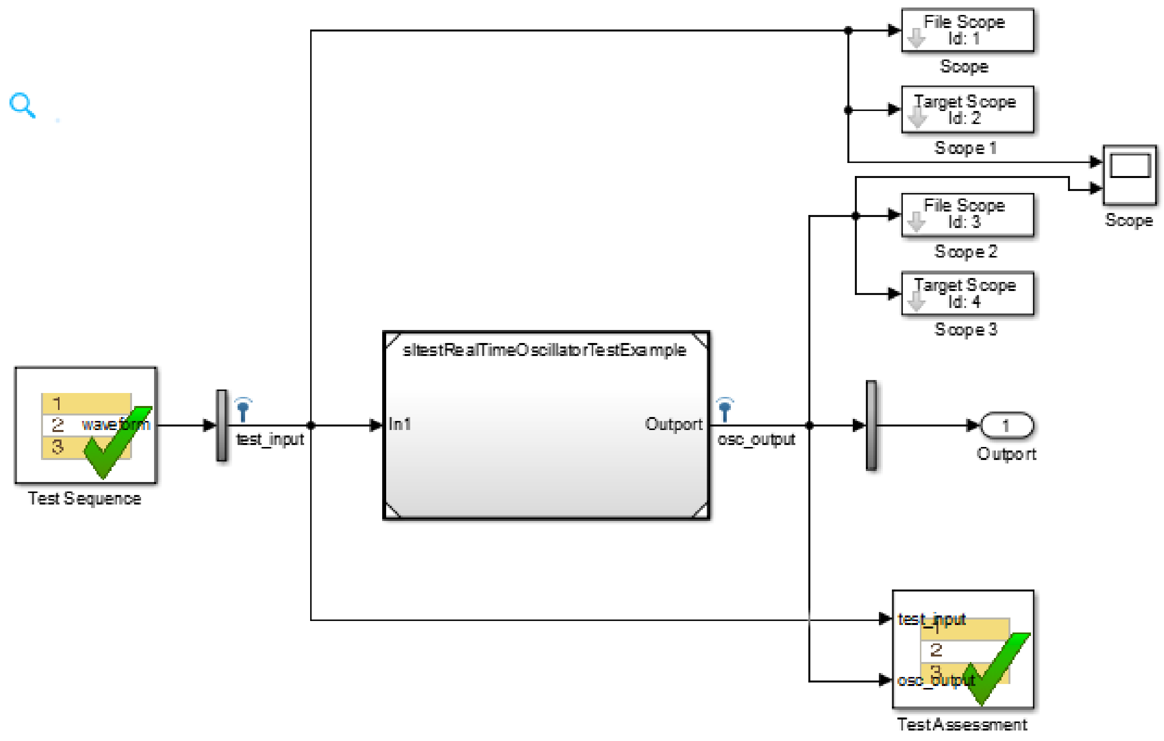
Real-time tests offer a **Pre-start real-time application** callback which executes commands just before the application executes on the target computer. Real-time test callbacks execute in a sequence along with the model load, build, download, and execute steps. Callbacks and step execution depends on how the test case loads the application.

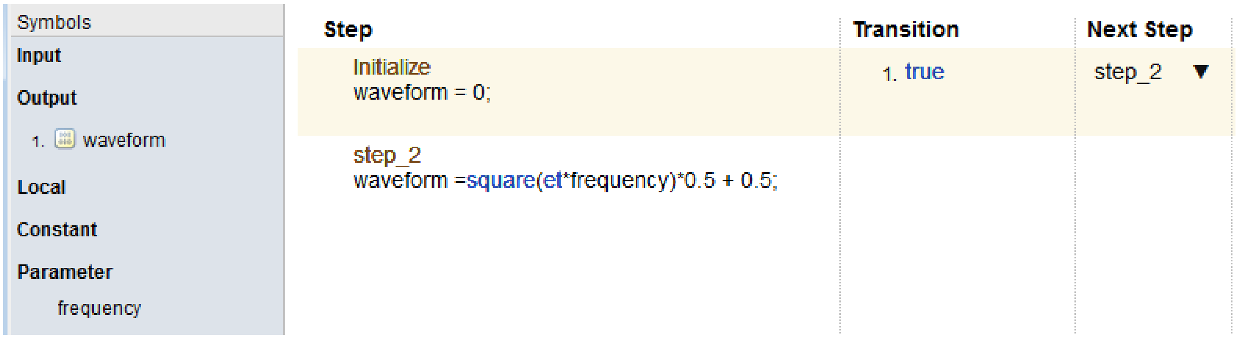
Sequence	Load application from: Model	Load application from: Target application	Load application from: Target computer
Executes first	**Preload callback**	**Preload callback**	**Preload callback**
	Post-load callback	—	—
	Pre-start real-time callback	**Pre-start real-time callback**	**Pre-start real-time callback**
Executes last	**Cleanup callback**	**Cleanup callback**	**Cleanup callback**

Iterations

You can execute iterations in real-time tests. Iterations are convenient for executing real-time tests that sweep through parameter values or Signal Builder groups. Results appear grouped by iteration. For more information on setting up iterations, see “Test Iterations” on page 7-71. You can create:

- Tabled iterations from a parameter set — Define several parameter sets in the **Parameter Overrides** section of the test case. Under **Iterations > Table Iterations**, click **Auto Generate** and select **Parameter Set**.
- Tabled iterations from signal builder groups — If your model or test harness uses a signal builder input, under **Iterations > Table Iterations**, click **Auto Generate** and select **Signal Builder Group**. If you use a signal builder group, load the application from the model.
- Scripted iterations — Use scripts to iterate using model variables or parameters. For example, in the model `sltestRealTimeOscillatorTestExample`, the `SettlingTest` harness uses a Test Sequence block to create a square wave test signal for the oscillator system using the parameter `frequency`.





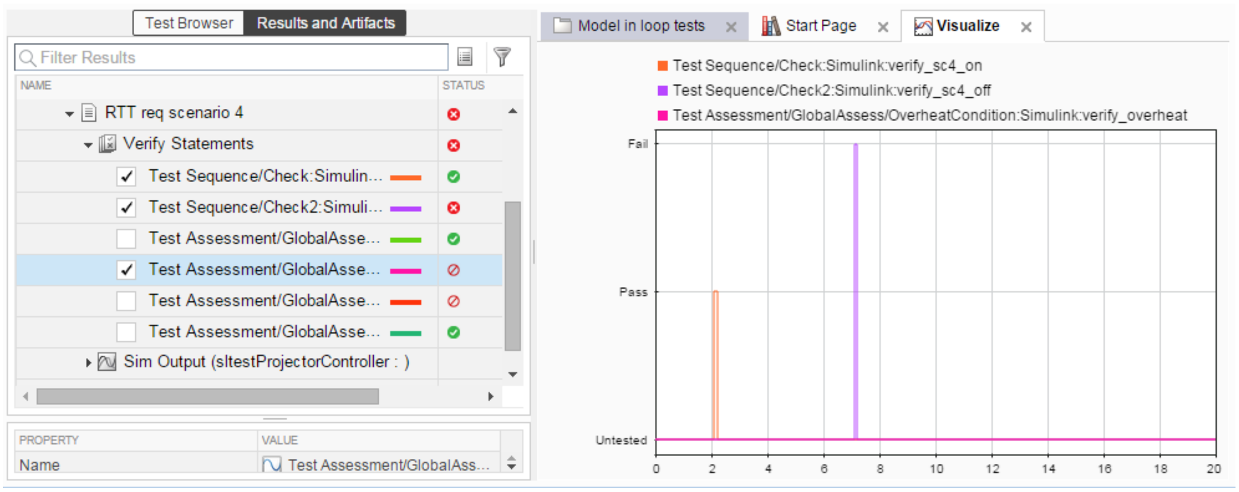
Step	Transition	Next Step
Initialize waveform = 0;	1. true	step_2 ▼
step_2 waveform =square(et*frequency)*0.5 + 0.5;		

In the test file `SettlingTestCases`, the real-time test scripted iterations cover a frequency sweep from 5 Hz to 35 Hz. The script iterates the value of `frequency` in the Test Sequence block.

```
%% Iterate over frequencies to determine best oscillator settings

% Create parameter sets
freq = 5.0:1.0:35.0;

for i_iter = 1:length(freq)
    % Create iteration object
    testItr = sltestiteration();

    % Set parameters
    setVariable(testItr,'Name','frequency','Source','Test Sequence',...
    'Value',freq(i_iter));

    % Register iteration
    addIteration(sltest_testCase, testItr);
end
```

Assess Real-Time Execution Using verify Statements

In addition to baseline and equivalence signal comparisons, you can assess real-time test execution using `verify` statements. A `verify` statement assesses a logical expression and returns results to the Test Manager. Use `verify` inside a Test Sequence or Test Assessment block. See “Assess Model Simulation Using verify Statements” on page 3-15.


Test Browser
Results and Artifacts
Filter Results
NAME
STATUS
RTT req scenario 4
Verify Statements
Test Sequence/Check:Simulin...
Test Sequence/Check2:Simuli...
Test Assessment/GlobalAsse...
Test Assessment/GlobalAsse...
Test Assessment/GlobalAsse...
Test Assessment/GlobalAsse...
Sim Output (sltestProjectorController :)
PROPERTY
VALUE
Name
Test Assessment/GlobalAss...
Model in loop tests
Start Page
Visualize
Test Sequence/Check:Simulink:verify_sc4_on
Test Sequence/Check2:Simulink:verify_sc4_off
Test Assessment/GlobalAssess/OverheatCondition:Simulink:verify_overheat
Fail
Pass
Untested
0
2
4
6
8
10
12
14
16
18
20

See Also

Related Examples

- “Test Real-Time Application” (Simulink Real-Time)

Reuse Desktop Test Cases for Real-Time Testing

Convert Desktop Test Cases to Real-Time

In the Test Manager, you can reuse test cases for real-time testing by converting desktop test cases to real-time test cases. For convenience, data can be stored externally so that each test case accesses common inputs and baseline data. The overall workflow is as follows:

1. Create a baseline, equivalence, or simulation test case with external inputs. For baseline tests, add baseline data from external files.
2. In the Test Manager, select the test case in the **Test Browser**.
3. Copy the test case. Right-click the test case and select **Copy**.
4. Paste the new test case into a test suite.
5. Rename the new test case.
6. Right-click the new test case, and select **Convert to** > **Real-Time Test**. For equivalence tests, select which simulation (simulation 1 or simulation 2) to run in real time.
7. Select the **Target Computer** and **Load Application From** options.
8. Ensure that the model settings are compatible with real-time test execution. For more information, see “Development Computer Setup and Configuration” (Simulink Real-Time).

Use External Data for Real-Time Tests

You can simplify test input data management by defining the input data in an external MAT or Excel file. Map the data to root inports in your model or test harness for desktop simulation. When you convert the desktop simulation test case into a real-time test, the test case uses the same inport mapping.

Using external data depends on how your test case loads the real-time application:

Load Real-Time Application from Model

If you are using external data for a real-time test, loading the real-time application from the model gives you the option of using an Excel file, MAT file, or CSV file. The external data is built into the application, and you can rerun the application from the target application or target computer.

In the **System Under Test** section, set the application to load from `Model`. In the **Inputs** section of the test case, click **Add**, and select an Excel file, MAT file, or CSV file. Map the data to your model inports. For more information on input mapping, see “Run Tests Using External Data” on page 7-25.

Load Real-Time Application from Target Application or Target Computer

After running the test from the model, you can run the test from the target application or target computer without recompiling. The application uses the input mapping from when the test ran from the model.

You can map external data to a test case loaded from the target application or target computer, without first running from the model. The external data must be in a MAT file, in the same format used if the test is loaded from the model. In the **System Under Test** section, select to load the application from the `Target Application` or `Target Computer`. In the **Inputs** section, click **Add** and select a MAT file. The Input string is not editable.

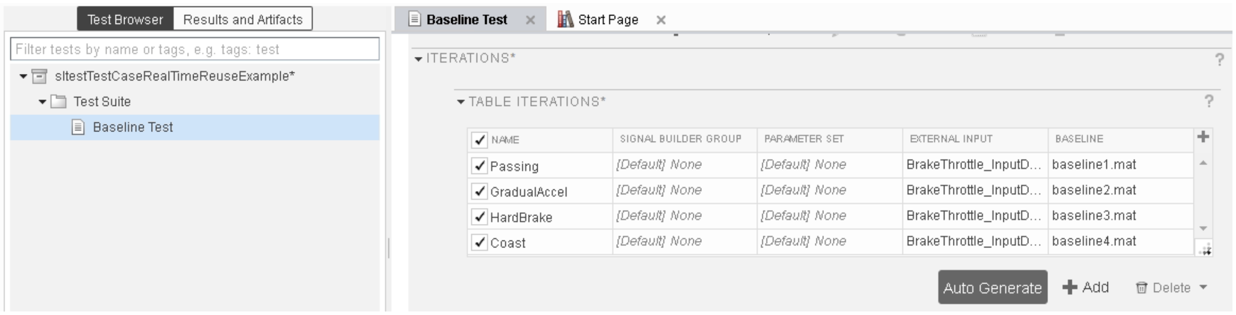
Example

This example shows a basic desktop test case reuse workflow using external input data defined in an Excel file. You run the baseline test case on the desktop, update the baseline data, convert a copy of the test case to a real-time test, then run the test case on a target computer. This example runs only on Windows systems.

1 Open the test file.

   ```
   tf = sltest.testmanager.TestFile(fullfile(matlabroot,'examples',...
   'simulinktest','sltestTestCaseRealTimeReuseExample.mldatx'));
   sltest.testmanager.load(tf.Name);
   sltest.testmanager.view;
   ```

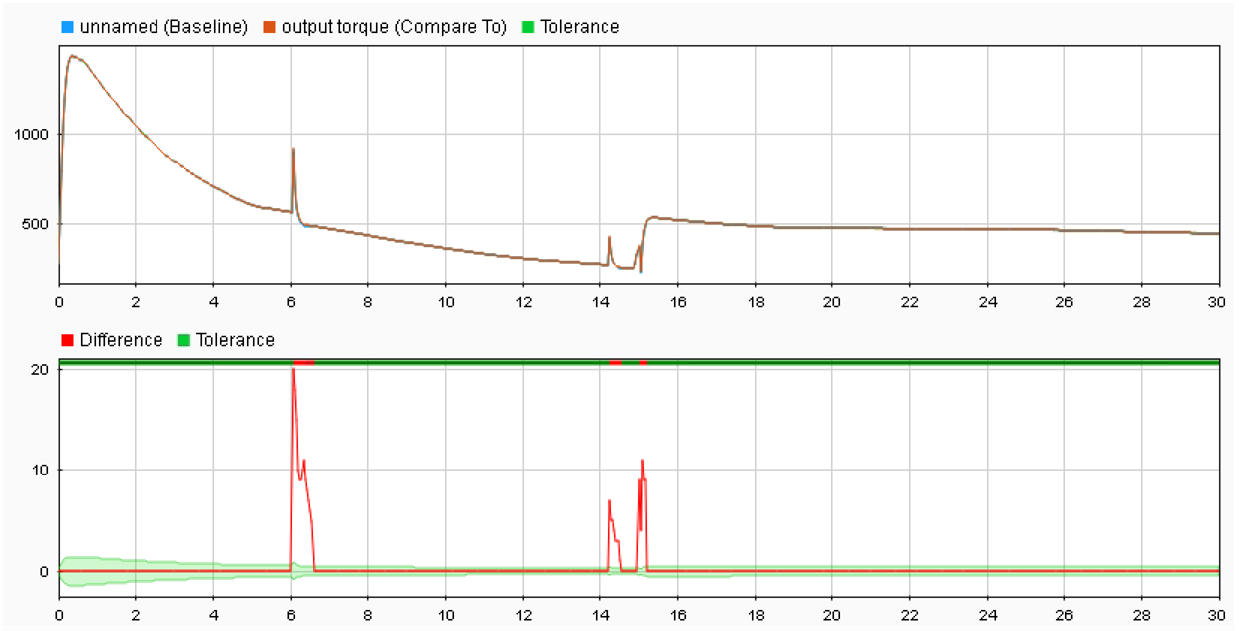
   The test file runs a transmission shift controller algorithm through four iterations, each corresponding to a different test scenario: passing, gradual acceleration, hard braking, and coasting. There is baseline data associated with each scenario for the signals `vehicle speed` and `output torque`.



✓ NAME	SIGNAL BUILDER GROUP	PARAMETER SET	EXTERNAL INPUT	BASELINE
✓ Passing	[Default] None	[Default] None	BrakeThrottle_InputD...	baseline1.mat
✓ GradualAccel	[Default] None	[Default] None	BrakeThrottle_InputD...	baseline2.mat
✓ HardBrake	[Default] None	[Default] None	BrakeThrottle_InputD...	baseline3.mat
✓ Coast	[Default] None	[Default] None	BrakeThrottle_InputD...	baseline4.mat

2 Run the baseline test.

3 Under the Baseline Criteria result, select `output torque` under the `Passing` result to view the comparison. The `Passing` result fails due to transient signals that fall outside the relative tolerance.



4 Assume that these transient signals are not significant, and update the baseline data:

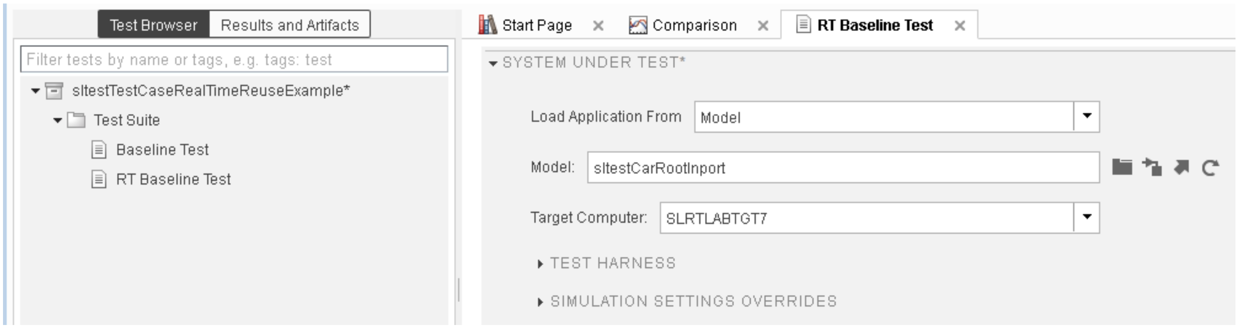
   1 Click **Next Failure**. The first failure region is bounded by data cursors.

   2 Click **Update Baseline** + **Update selected signal region**, and confirm that you want to overwrite the data.

   3 Repeat this process for the other two failure regions.
5 Copy and convert the baseline test case to a real-time test:

   1 In the Test Browser, right-click Baseline Test and select Copy.
   2 Paste the new test case under the test suite.
   3 Rename the new tests case `RT Baseline Test`.
   4 Right-click `RT Baseline Test` and select **Convert to** > **Real-Time Test**.
6 Run the real-time test case:

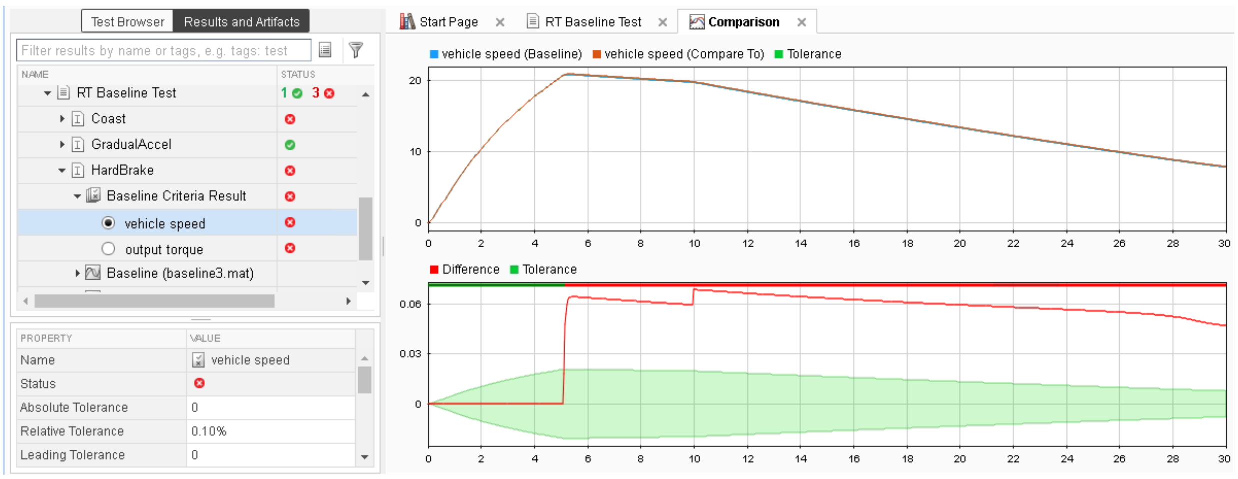
   1 Set the **Target Computer**.
   2 Set the system under test to load from `Model`.



Load Application From: Model
Model: sltestCarRootInport
Target Computer: SLRTLABTGT7

   3 Run the `RT Baseline Test` test case.
7 In this example, several of the scenarios fail due to timing impacts on the data output. For example, in the `HardBrake` iteration, the `vehicle speed` output falls outside the relative tolerance after the brake is applied. To resolve this failure, you could:

   - Increase the relative tolerance for the real-time test.
   - Create a separate set of baseline data for the real-time test.



NAME	STATUS
RT Baseline Test	1 ✓ 3 ✗
Coast	✗
GradualAccel	✓
HardBrake	✗
Baseline Criteria Result	✗
vehicle speed	✗
output torque	✗
Baseline (baseline3.mat)	

PROPERTY	VALUE
Name	vehicle speed
Status	✗
Absolute Tolerance	0
Relative Tolerance	0.10%
Leading Tolerance	0

See Also

Related Examples

- “Test Real-Time Application” (Simulink Real-Time)

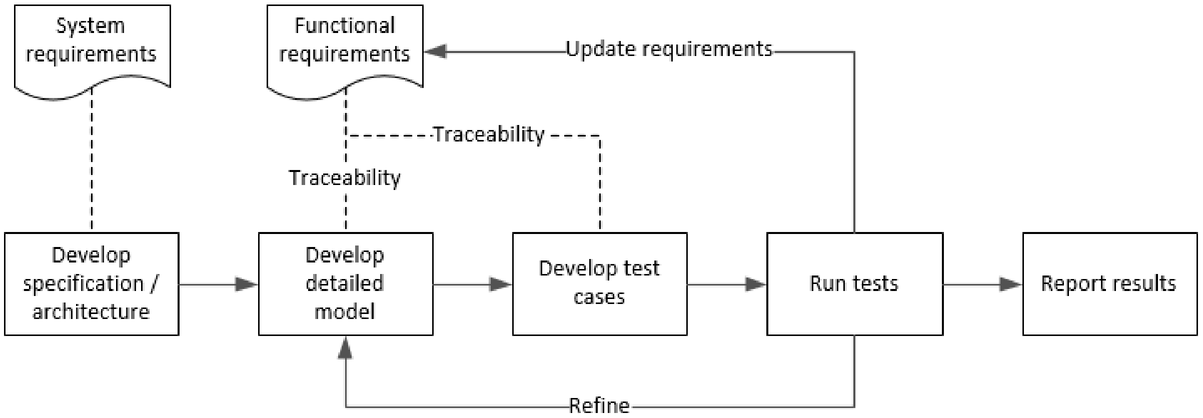
Verification and Validation

Test Model Against Requirements and Report Results

Requirements – Test Traceability Overview

Traceability between requirements and test cases helps you interpret test results and see the extent to which your requirements are verified. You can link a requirement to elements that help verify it, such as test cases in the Test Manager, `verify` statements in a Test Sequence block, or Model Verification blocks in a model. When you run tests, a pass/fail summary appears in your requirements set.

This example demonstrates a common requirements-based testing workflow for a cruise control model. You start with a requirements set, a model, and a test case. You add traceability between the tests and the safety requirements. You run the test, summarize the verification status, and report the results.



In this example, you conduct a simple test of two requirements in the set:

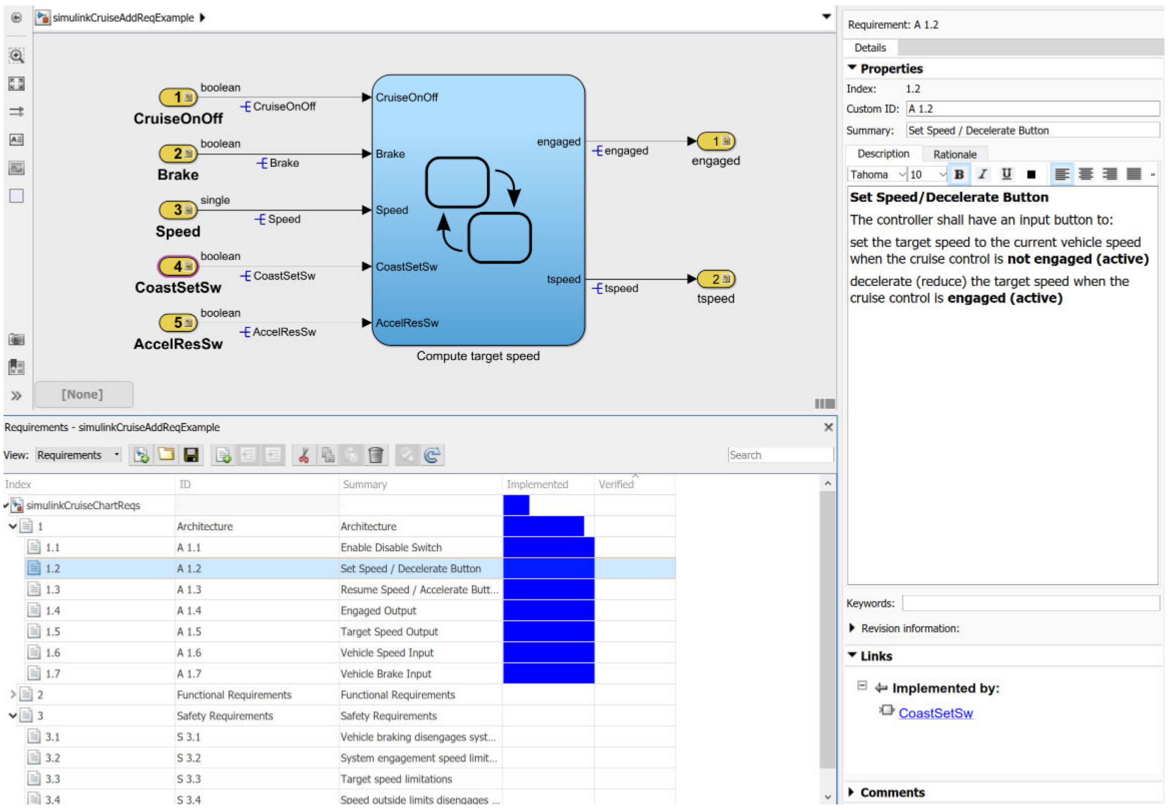
- That the cruise control system transitions to disengaged from engaged when a braking event has occurred
- That the cruise control system transitions to disengaged from engaged when the current vehicle speed is outside the range of 20 mph to 90 mph.

Display the Requirements and Test Case

1. Create a copy of the project in a working folder. The project contains data, documents, models, and tests. Enter:

   ```
   path = fullfile(matlabroot,'toolbox','shared','examples',...
   'verification','src','cruise')
   run(fullfile(path,'slVerificationCruiseStart'))
   ```

2. In the project `models` folder, open the `simulinkCruiseAddReqExample.slx` model.
3. Display the requirements. Click the icon in the lower-right corner of the model canvas, and select **Requirements**. The requirements appear below the model canvas.
4. Expand the requirements information to include verification and implementation status. Right-click a requirement and select **Verification Status** and **Implementation Status**.



Requirements - simulinkCruiseAddReqExample

View: Requirements

Search

Index	ID	Summary	Implemented	Verified
simulinkCruiseChartReqs			🟦	
1	Architecture	Architecture	🟦	
1.1	A 1.1	Enable Disable Switch	🟦	
1.2	A 1.2	Set Speed / Decelerate Button	🟦	
1.3	A 1.3	Resume Speed / Accelerate Butt...	🟦	
1.4	A 1.4	Engaged Output	🟦	
1.5	A 1.5	Target Speed Output	🟦	
1.6	A 1.6	Vehicle Speed Input	🟦	
1.7	A 1.7	Vehicle Brake Input	🟦	
2	Functional Requirements	Functional Requirements		
3	Safety Requirements	Safety Requirements		
3.1	S 3.1	Vehicle braking disengages syst...		
3.2	S 3.2	System engagement speed limit...		
3.3	S 3.3	Target speed limitations		
3.4	S 3.4	Speed outside limits disengages ...		

Requirement: A 1.2

Details

Properties

Index: 1.2

Custom ID: A 1.2

Summary: Set Speed / Decelerate Button

Description | Rationale

Set Speed/Decelerate Button

The controller shall have an input button to:

set the target speed to the current vehicle speed when the cruise control is **not engaged (active)**

decelerate (reduce) the target speed when the cruise control is **engaged (active)**

Keywords:

Revision information:

Links

Implemented by:

CoastSetSw

Comments

5 Open the Simulink Test file `slReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager.

Link Requirements to Tests

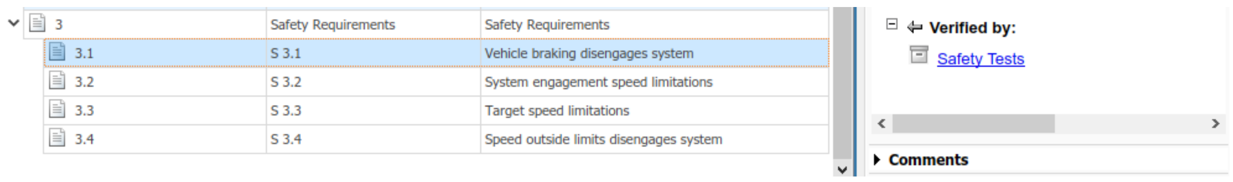
Link the requirements to the test case.

1 In the Requirements Browser, select requirement `S 3.1`.

2 In the Test Manager, expand the test file and select the **Safety Tests** test case. Expand the **Requirements** section.

3 In the **Requirements** section, select **Add > Link to Selected Requirement**.

The requirements browser displays the verification-type link.



3	Safety Requirements	Safety Requirements
3.1	S 3.1	Vehicle braking disengages system
3.2	S 3.2	System engagement speed limitations
3.3	S 3.3	Target speed limitations
3.4	S 3.4	Speed outside limits disengages system

Verified by:

Safety Tests

Comments

4 Also add a link for item `S 3.4`.

Run the Test

1 The test case uses a test harness `SafetyTest_Harness1`. In the test harness, a test sequence sets the input conditions and checks the model behavior:

- The `BrakeTest` sequence engages the cruise control, then applies the brake. It includes the `verify` statement

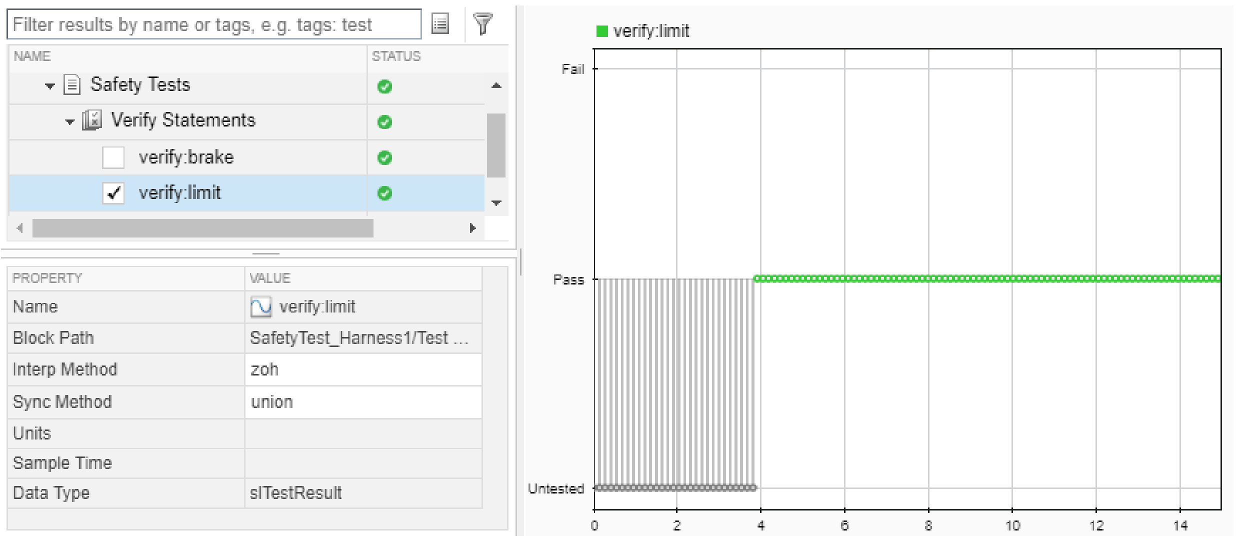
```
verify(engaged == false,...
    'verify:brake',...
    'system must disengage when brake applied')
```

- The `LimitTest` sequence engages the cruise control, then ramps up the vehicle speed until it exceeds the upper limit. It includes the `verify` statement.

```
verify(engaged == false,...
    'verify:limit',...
    'system must disengage when limit exceeded')
```

2 Run the test case. In the Test Manager toolstrip, click **Run**.

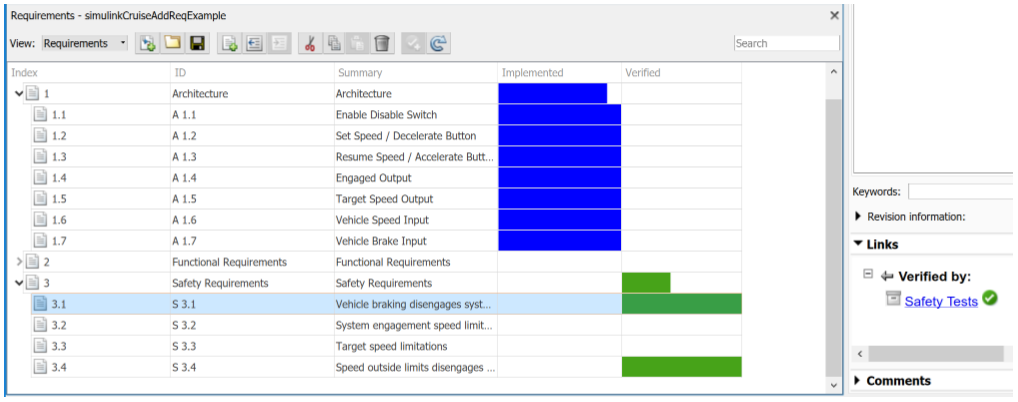
3 When the test finishes, expand the **Verify Statements** results. The Test Manager results show that both assessments pass, and the plot shows the detailed results of each `verify` statement.


Filter results by name or tags, e.g. tags: test

NAME	STATUS
Safety Tests	✓
Verify Statements	✓
☐ verify:brake	✓
☑ verify:limit	✓

PROPERTY	VALUE
Name	verify:limit
Block Path	SafetyTest_Harness1/Test ...
Interp Method	zoh
Sync Method	union
Units	
Sample Time	
Data Type	slTestResult

4 In the Requirements Browser, right-click a requirement and select **Refresh Verification Status** to show the passing test results for each requirement.


Requirements - simulinkCruiseAddReqExample

View: Requirements

Index	ID	Summary	Implemented	Verified
1	Architecture	Architecture	🟦	
1.1	A 1.1	Enable Disable Switch	🟦	
1.2	A 1.2	Set Speed / Decelerate Button	🟦	
1.3	A 1.3	Resume Speed / Accelerate Butt...	🟦	
1.4	A 1.4	Engaged Output	🟦	
1.5	A 1.5	Target Speed Output	🟦	
1.6	A 1.6	Vehicle Speed Input	🟦	
1.7	A 1.7	Vehicle Brake Input	🟦	
2	Functional Requirements	Functional Requirements		
3	Safety Requirements	Safety Requirements		🟩
3.1	S 3.1	Vehicle braking disengages syst...		🟩
3.2	S 3.2	System engagement speed limit...		
3.3	S 3.3	Target speed limitations		
3.4	S 3.4	Speed outside limits disengages ...		🟩

Keywords:

Revision information:

Links

Verified by:

Safety Tests ✓

Comments

Report the Results

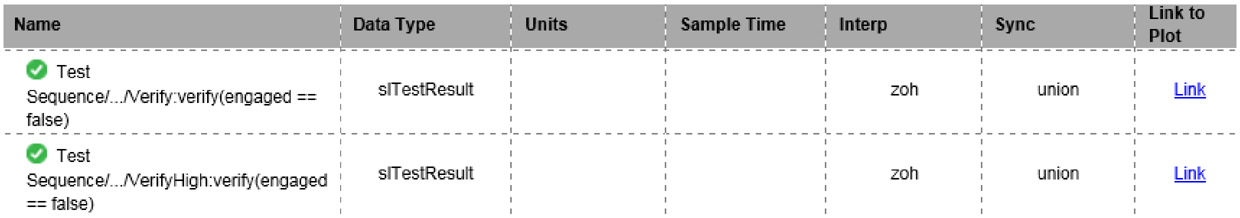
1 Create a report using a custom Microsoft Word template.

a From the Test Manager results, right-click the test case name. Select **Create Report**.

- **b** In the Create Test Result Report dialog box, set the options:
  - Title — `SafetyTest`
  - Results for — `All Tests`
  - File Format — `DOCX`
  - For the other options, keep the default selections.
- **c** For the **Template File**, select the `ReportTemplate.dotx` file in the **documents** project folder.
- **d** Enter a file name and select a location for the report.
- **e** Click **Create**.

2 Review the report.

- **a** In the **Test Case Requirements** section, click the link to trace to the requirements document.
- **b** The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.



Name	Data Type	Units	Sample Time	Interp	Sync	Link to Plot
✓ Test Sequence/.../Verify:verify(engaged == false)	slTestResult			zoh	union	Link
✓ Test Sequence/.../VerifyHigh:verify(engaged == false)	slTestResult			zoh	union	Link

See Also

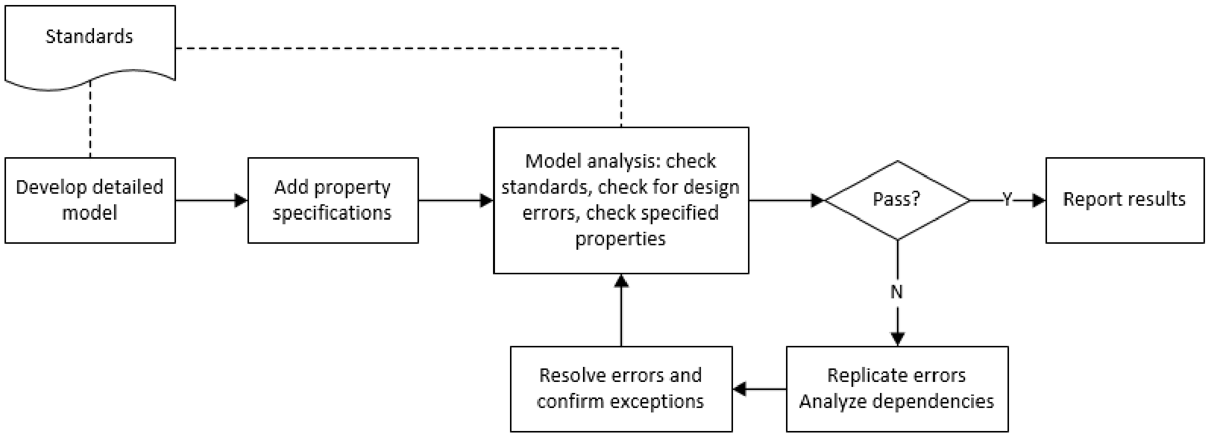
Related Examples

- “Link to Requirements” on page 1-2
- “Validate Requirements Links in a Model” (Simulink Requirements)
- “Customize Requirements Traceability Report for Model” (Simulink Requirements)

Analyze a Model for Standards Compliance and Design Errors

Standards and Analysis Overview

During model development, check and analyze your model to increase confidence in its quality. Check your model against standards such as MAAB style guidelines and high-integrity system design guidelines such as DO-178 and ISO 26262. Analyze your model for errors, dead logic, and conditions that violate required properties. Using the analysis results, update your model and document exceptions. Report the results using customizable templates.


Standards
Develop detailed model
Add property specifications
Model analysis: check standards, check for design errors, check specified properties
Pass?
Y
Report results
N
Replicate errors Analyze dependencies
Resolve errors and confirm exceptions

Check Model for Style Guideline Violations and Design Errors

This example shows how to use the Model Advisor to check a cruise control model for MathWorks® Automotive Advisory Board (MAAB) style guideline violations and design errors. Select checks and run the analysis on the model. Iteratively debug issues using the Model Advisor and rerun checks to verify that it is in compliance. After passing your selected checks, report results.

Check Model for MAAB Style Guideline Violations

In Model Advisor, you can check that your model complies with MAAB modeling guidelines.

1. Create a copy of the project in a working folder. On the command line, enter

   ```
   path = fullfile(matlabroot,'toolbox','shared','examples',...
   'verification','src','cruise')
   run(fullfile(path,'slVerificationCruiseStart'))
   ```
2. Open the model. On the command line, enter

   ```
   open_system simulinkCruiseErrorAndStandardsExample
   ```
3. In the model window, select **Analysis** > **Model Advisor** > **Model Advisor**.
4. Click OK to choose `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.
5. Check your model for MAAB style guideline violations using Simulink Check.

   a. In the left pane, in the **By Product > Simulink Check > Modeling Standards > MathWorks Automotive Advisory Board Checks** folder, select:

      - **Check for indexing in blocks**
      - **Check for prohibited blocks in discrete controllers**
      - **Check model diagnostic parameters**

   b. Right-click the **MathWorks Automotive Advisory Board Checks** node, and then select `Run Selected Checks`.

   c. Click **Check model diagnostic parameters** to review the configuration parameter settings that violate MAAB style guidelines.

   d. In the right pane, click the parameter links to update the values in the Configuration Parameters dialog box.

   e. To verify that your model passes, rerun the check. Repeat steps `c` and `d`, if necessary, to reach compliance.

   f. To generate a results report of the Simulink Check checks, select the **MathWorks Automotive Advisory Board Checks** node, and then, in the right pane click **Generate Report...**.

Check Model for Design Errors

While in Model Advisor, you can also check your model for hidden design errors using Simulink Design Verifier.

1. In the left pane, in the **By Product > Simulink Design Verifier** folder, select **Design Error Detection**.
2. In the right pane, click **Run Selected Checks**.
3. After the analysis is complete, expand the **Design Error Detection** folder, then select checks to review warnings or errors.
4. In the right pane, click **Simulink Design Verifier Results Summary**. The dialog box provides tools to help you diagnose errors and warnings in your model.

   a. Review the results on the model. Click **Highlight analysis results on model**. Click the `Compute target speed` subsystem, outlined in red. The Simulink Design Verifier Results Inspector window provides derived ranges that can help you understand the source of an error by identifying the possible signal values.

   b. Review the harness model. The Simulink Design Verifier Results Inspector window displays information that an overflow error occurred. To see the test cases that demonstrate the errors, click **View test case**.

   c. Review the analysis report. In the Simulink Design Verifier Results Inspector window, click **Back to summary**. To see a detailed analysis report, click **HTML** or **PDF**.

See Also

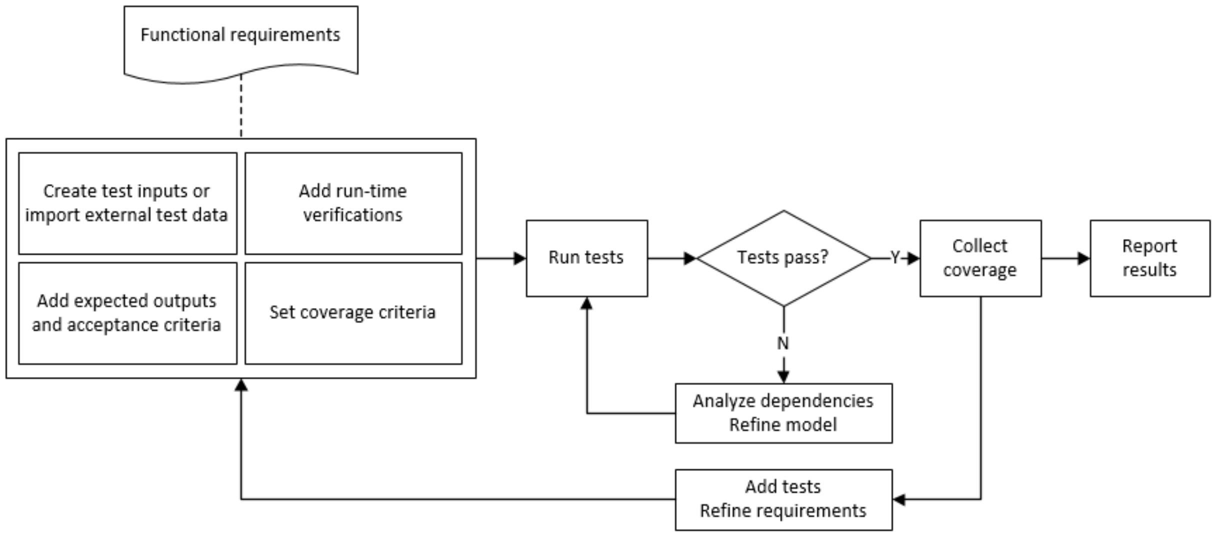
Related Examples

- “Check Model Compliance by Using the Model Advisor” (Simulink Check)
- “Collect Model Metrics Using the Model Advisor” (Simulink Check)
- “Run a Design Error Detection Analysis” (Simulink Design Verifier)
- “Prove Properties in a Model” (Simulink Design Verifier)

Perform Functional Testing and Analyze Test Coverage

Functional testing begins with building test cases based on requirements. These tests can cover key aspects of your design and verify that individual model components meet requirements. Test cases include inputs, expected outputs, and acceptance criteria.

By collecting individual test cases within test suites, you can run functional tests systematically. To check for regression, add baseline criteria to the test cases and test the model iteratively. Coverage measurement reflects the extent to which these tests have fully exercised the model. Coverage measurement also helps you to add tests and requirements to meet coverage targets.


Functional requirements
Create test inputs or import external test data
Add run-time verifications
Add expected outputs and acceptance criteria
Set coverage criteria
Run tests
Tests pass?
Y
N
Collect coverage
Report results
Analyze dependencies Refine model
Add tests Refine requirements

Incrementally Increase Test Coverage Using Test Case Generation

This example shows a functional testing-based testing workflow for a cruise control model. You start with a model that has tests linked to an external requirements document, analyze the model for coverage in Simulink Coverage, incrementally increase coverage with Simulink Design Verifier, and report the results.

Explore the Test Harness and the Model

1. Create a copy of the project in a working folder. At the command line, enter:

   ```
   path = fullfile(matlabroot,'toolbox','shared','examples',...
   'verification','src','cruise')
   run(fullfile(path,'slVerificationCruiseStart'))
   ```
2. Open the model and the test harness. At the command line, enter:

   ```
   open_system simulinkCruiseAddReqExample
   sltest.harness.open('simulinkCruiseAddReqExample','SafetyTest_Harness1')
   ```
3. Load the test suite from “Test Model Against Requirements and Report Results” on page 10-2. At the command line, enter:

   ```
   sltest.testmanager.load('slReqTests.mldatx')
   sltest.testmanager.view
   ```
4. Open the test sequence block. The sequence tests:

   - That the system disengages when the brake pedal is pressed
   - That the system disengages when the speed exceeds a limit

   Some test sequence steps are linked to a requirements document `simulinkCruiseChartReqs.docx`.

Measure Model Coverage

1. In the Test Manager, enable coverage collection for the test case.

   a. Open the Test Manager. In the Simulink menu, click **Analysis > Test Manager**.

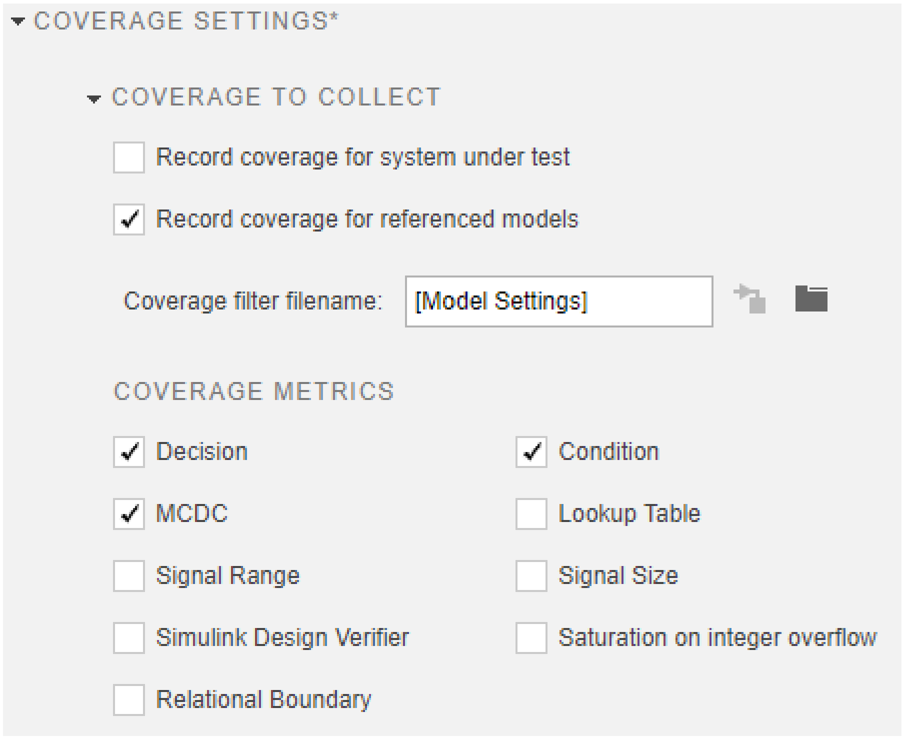
   b. In the **Test Browser**, click the `slReqTests` test file.

   c. Expand **Coverage Settings**.

   d. Under **Coverage to Collect**, select **Record coverage for referenced models**.

      You specify a coverage filter to use for coverage analysis by using the **Coverage filter filename** field. The default setting honors the model configuration parameter settings. Leaving the **Coverage filter filename** field empty attaches no coverage filter.

   e. Under **Coverage Metrics**, select **Decision**, **Condition**, and **MCDC**.


COVERAGE SETTINGS*

COVERAGE TO COLLECT

- [ ] Record coverage for system under test
- [✓] Record coverage for referenced models

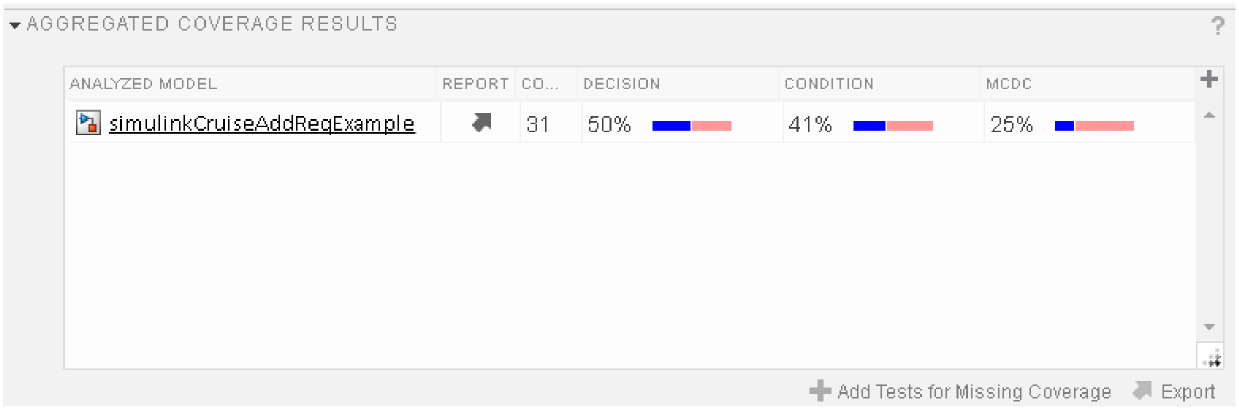
Coverage filter filename: [Model Settings]

COVERAGE METRICS

- [✓] Decision
- [✓] Condition
- [✓] MCDC
- [ ] Lookup Table
- [ ] Signal Range
- [ ] Signal Size
- [ ] Simulink Design Verifier
- [ ] Saturation on integer overflow
- [ ] Relational Boundary

2 Run the test. On the Test Manager toolstrip, click **Run**.

3 When the test finishes, in the Test Manager, navigate to the test case. The aggregated coverage results show that the example model achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.



ANALYZED MODEL	REPORT	CO...	DECISION	CONDITION	MCDC
simulinkCruiseAddReqExample		31	50%	41%	25%

Generate Tests to Increase Model Coverage

1. Use Simulink Design Verifier to generate additional tests to increase model coverage. Select the test case in the **Results and Artifacts** and open the aggregated coverage results section.
2. Select the test results from the previous section and then click **Add Tests for Missing Coverage**.

   The **Add Tests for Missing Coverage** options open.
3. Under **Harness**, choose `Create a new harness`.
4. Click **OK** to add tests to the test suite using Simulink Design Verifier.
5. Run the updated test suite. On the Test Manager toolstrip, click **Run**. The test results include coverage for the combined test case inputs, achieving increased model coverage.

See Also

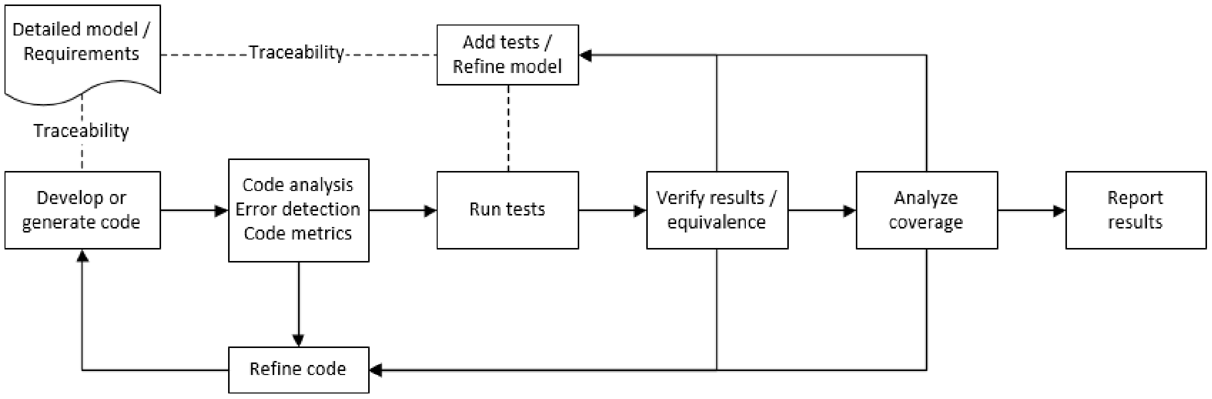
Related Examples

- “Link to Requirements” on page 1-2
- “Assess Model Simulation Using verify Statements” on page 3-15
- “Compare Model Output To Baseline Data” on page 7-9
- “Generate Test Cases for Model Decision Coverage” (Simulink Design Verifier)
- “Increase Test Coverage for a Model”

Analyze Code and Test Software-in-the-Loop

Code Analysis and Testing Software-in-the-Loop Overview

Analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. Typically for handwritten code, you check for run-time errors with static code analysis and run test cases that evaluate the code against requirements and evaluate code coverage. Based on the results, refine the code and add tests. For generated code, demonstrate that code execution produces equivalent results to the model by using the same test cases and baseline results. Compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.



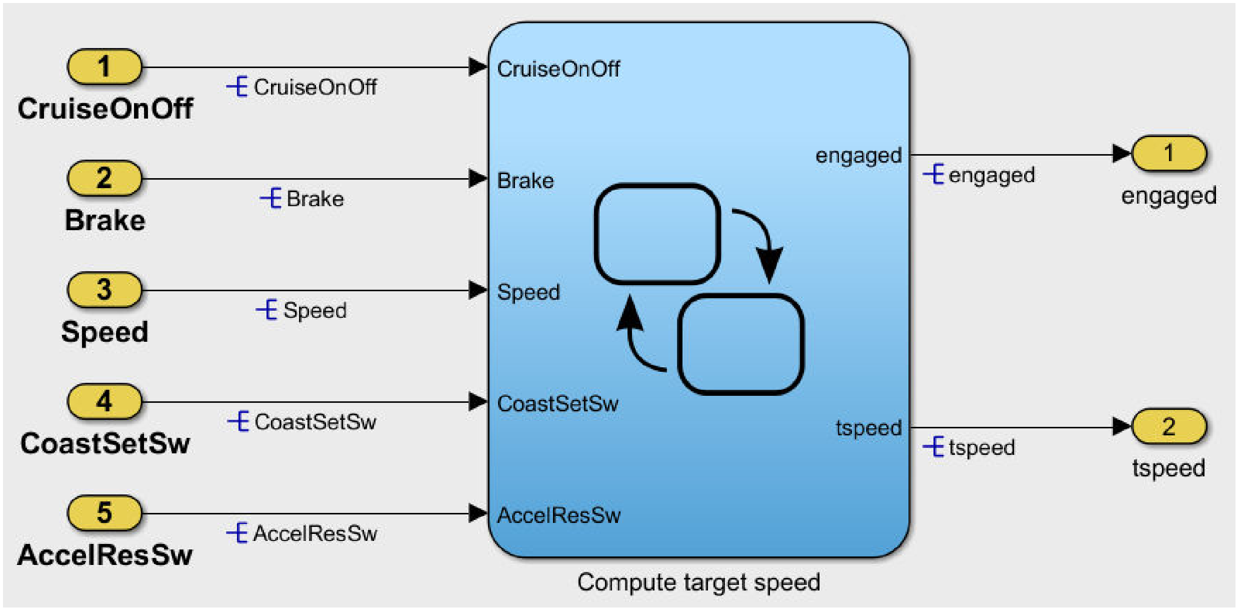
Analyze Code for Defects, Metrics, and MISRA C:2012

This workflow describes how to check if your model produces MISRA® C:2012 compliant code and how to check your generated code for code metrics, code defects, and MISRA compliance. To produce more MISRA compliant code from your model, you use the code generation and Model Advisor. To check whether the code is MISRA compliant, you use the Polyspace MISRA C:2012 checker and report generation capabilities. For this example, you use the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

1 Open the project.

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'slVerificationCruiseStart'))
```

2 From the project, open the model `simulinkCruiseErrorAndStandardsExample`.

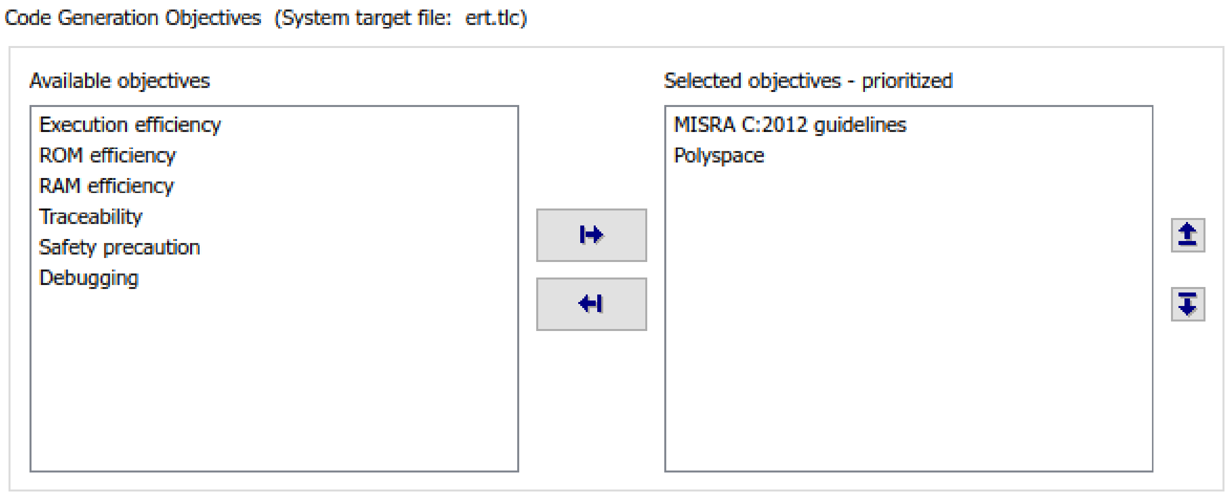


Run Code Generator Checks

Before you generate code from your model, there are steps that you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows how to use the Code Generation Advisor to check your model before generating code.

1 Right-click Compute target speed and select **C/C++** > **Code Generation Advisor**.

2 Select the Code Generation Advisor folder. Add the `Polyspace` objective. The `MISRA C:2012 guidelines` objective is already selected.

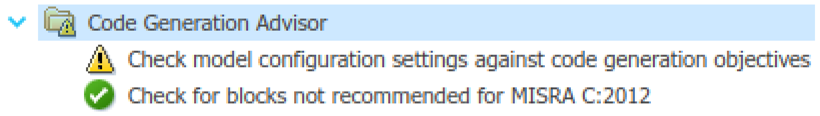

Code Generation Objectives (System target file: ert.tlc)

Available objectives:
- Execution efficiency
- ROM efficiency
- RAM efficiency
- Traceability
- Safety precaution
- Debugging

Selected objectives - prioritized:
- MISRA C:2012 guidelines
- Polyspace

3 Click **Run Selected Checks**.

The Code Generation Advisor checks whether there are any blocks or configuration settings that are not recommended for MISRA C:2012 compliance and Polyspace code analysis. For this mode, the check for incompatible blocks passes, but there are some configuration settings that are incompatible with MISRA compliance and Polyspace checking.


Code Generation Advisor
- ⚠ Check model configuration settings against code generation objectives
- ✓ Check for blocks not recommended for MISRA C:2012

4 Click on check that was not passed. Accept the parameter changes by selecting **Modify Parameters**.

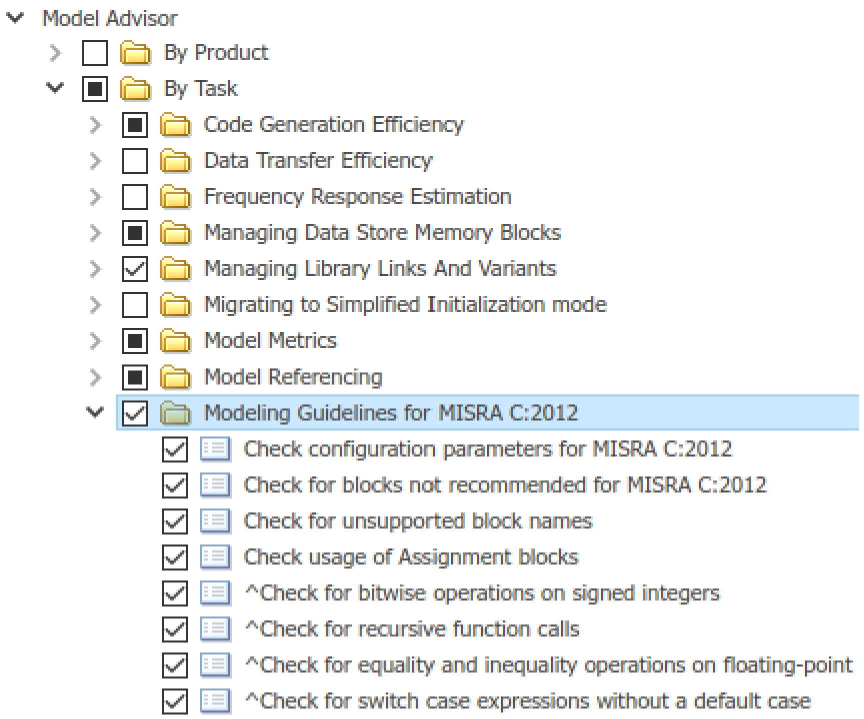
5 Rerun the check by selecting **Run This Check**.

Run Model Advisor Checks

Before you generate code from your model, there are steps you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows you how to use the Model Advisor to check your model further before generating code.

For more checking before generating code, you can also run the Modeling Guidelines for MISRA C:2012.

1. At the bottom of the Code Generation Advisor window, select **Model Advisor**.
2. Under the **By Task** folder, select the **Modeling Guidelines for MISRA C:2012** advisor checks.



Model Advisor
- ☐ By Product
- ■ By Task
  - ■ Code Generation Efficiency
  - ☐ Data Transfer Efficiency
  - ☐ Frequency Response Estimation
  - ■ Managing Data Store Memory Blocks
  - ☑ Managing Library Links And Variants
  - ☐ Migrating to Simplified Initialization mode
  - ■ Model Metrics
  - ■ Model Referencing
  - ☑ Modeling Guidelines for MISRA C:2012
    - ☑ Check configuration parameters for MISRA C:2012
    - ☑ Check for blocks not recommended for MISRA C:2012
    - ☑ Check for unsupported block names
    - ☑ Check usage of Assignment blocks
    - ☑ ^Check for bitwise operations on signed integers
    - ☑ ^Check for recursive function calls
    - ☑ ^Check for equality and inequality operations on floating-point
    - ☑ ^Check for switch case expressions without a default case

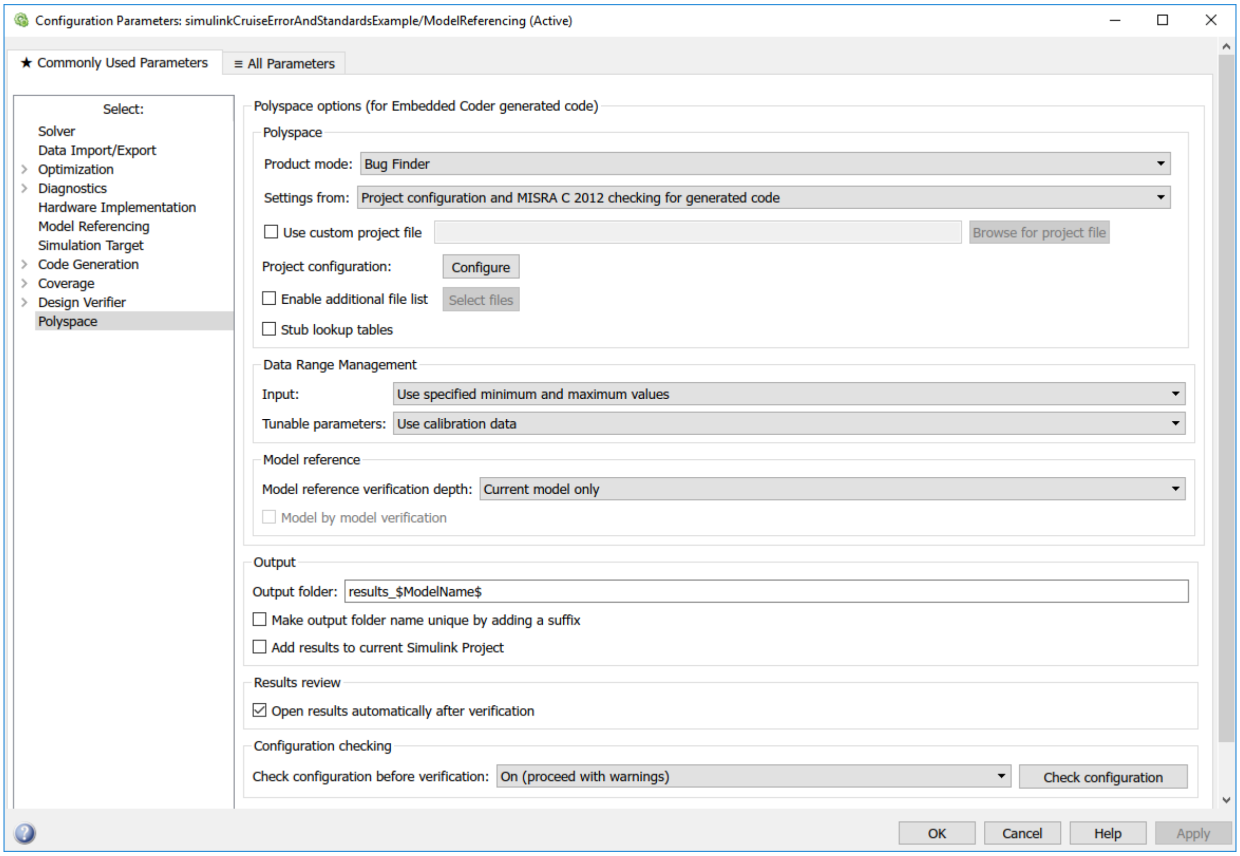
3. Click **Run Selected Checks** and review the results.
4. If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

Generate and Analyze Code

After you have done the model compliance checking, you can now generate code. With Polyspace, you can check your code for compliance with MISRA C:2012 and generate reports to demonstrate compliance with MISRA C:2012.

1. In the Simulink editor, right-click Compute target speed and select **C/C++ > Build This Subsystem**.
2. Use the default settings for the tunable parameters and select **Build**.

3 After the code is generated, right-click Compute target speed and select **Polyspace** > **Options**.


Configuration Parameters: simulinkCruiseErrorAndStandardsExample/ModelReferencing (Active)

Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: Project configuration and MISRA C 2012 checking for generated code

☐ Use custom project file [Browse for project file]

Project configuration: [Configure]

☐ Enable additional file list [Select files]

☐ Stub lookup tables

Data Range Management

Input: Use specified minimum and maximum values

Tunable parameters: Use calibration data

Model reference

Model reference verification depth: Current model only

☐ Model by model verification

Output

Output folder: results_$ModelName$

☐ Make output folder name unique by adding a suffix

☐ Add results to current Simulink Project

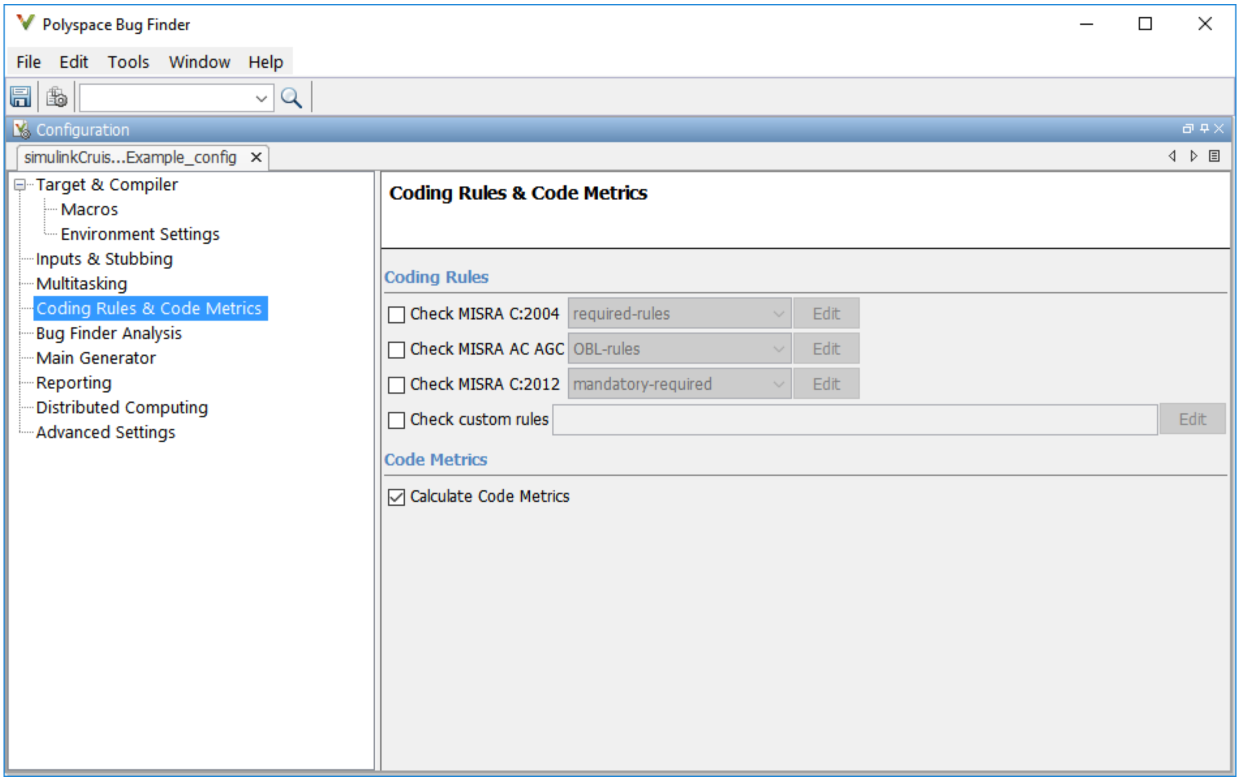
Results review

☑ Open results automatically after verification

Configuration checking

Check configuration before verification: On (proceed with warnings) [Check configuration]

4 Click the **Configure** (Polyspace Bug Finder) button. This option allows you to choose more advanced Polyspace analysis options in the Polyspace configuration window.



Coding Rules & Code Metrics

Coding Rules

- [ ] Check MISRA C:2004 — required-rules
- [ ] Check MISRA AC AGC — OBL-rules
- [ ] Check MISRA C:2012 — mandatory-required
- [ ] Check custom rules

Code Metrics

- [x] Calculate Code Metrics

5 On the same pane, select **Calculate Code Metrics**. This option turns on code metric calculations for your generated code.

6 Save and close the Polyspace configuration window.

7 From your model, right-click Compute target speed and select **Polyspace > Verify Code Generated For > Selected Subsystem**.

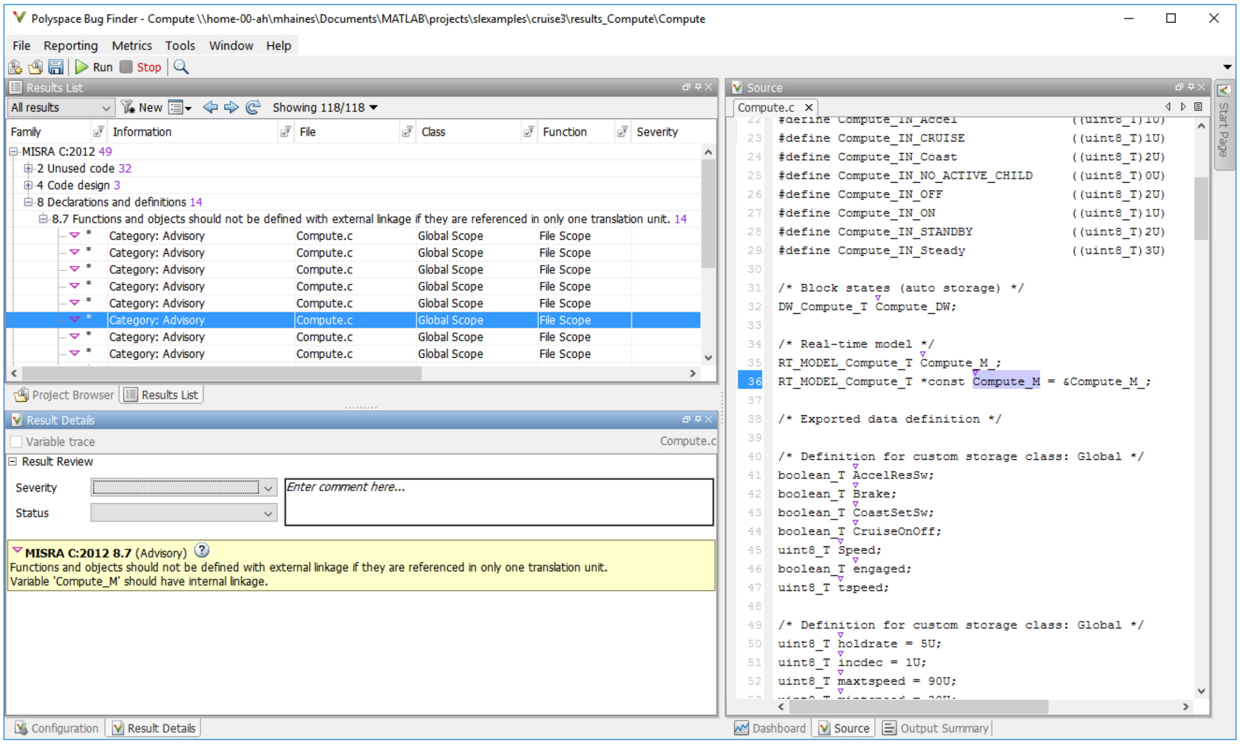
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

Review Results

After you run a Polyspace analysis of your generated code, the Polyspace environment shows you the results of the static code analysis. There are 50 MISRA C:2012 coding rule violations in your generated code.

1. Expand the tree for rule 8.7 and click through the different results.

   Rule 8.7 states that functions and objects should not be global if the function or object is local. As you click through the 8.7 violations, you can see that these results refer to variables that other components also use, such as `CruiseOnOff`. You can annotate your code or your model to justify every result. But, because this model is a unit in a larger program, you can also change the configuration of the analysis to check only a subset of MISRA rules.



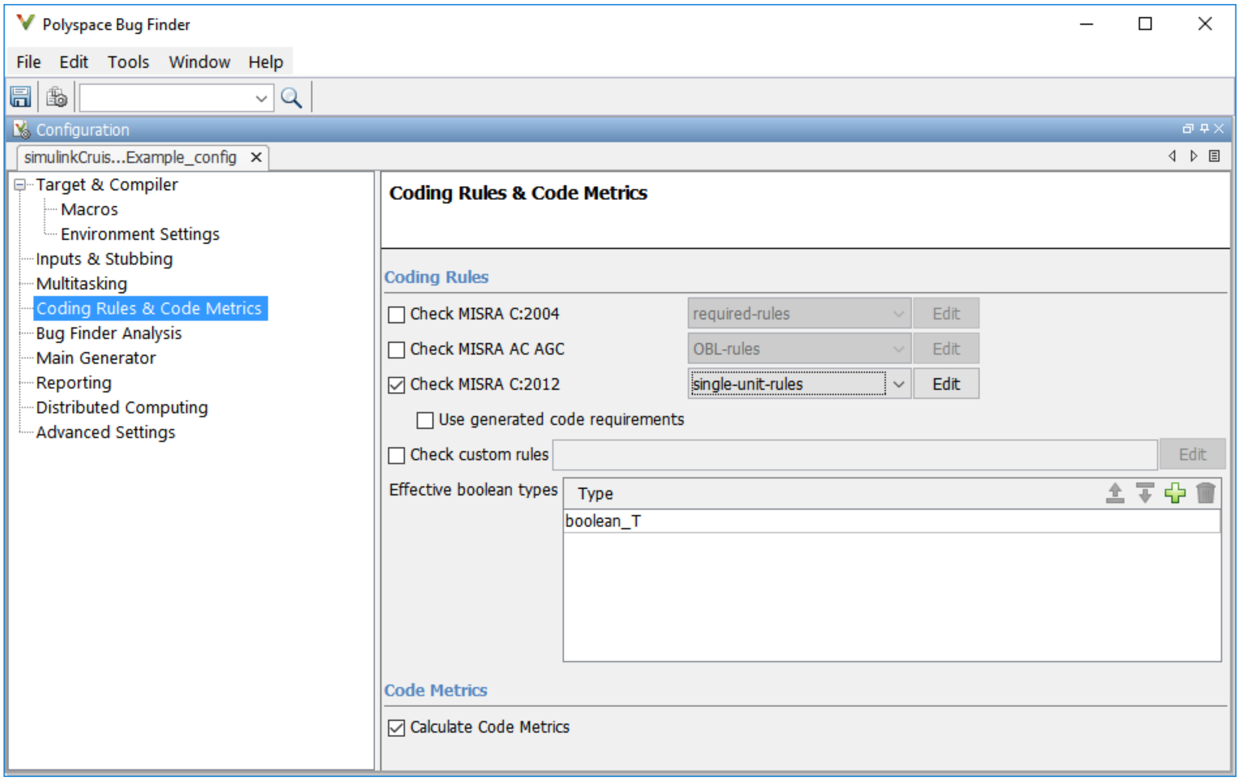
Family	Information	File	Class	Function	Severity
MISRA C:2012 49					
2 Unused code 32					
4 Code design 3					
8 Declarations and definitions 14					
8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit. 14					
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	
*	Category: Advisory	Compute.c	Global Scope	File Scope	

MISRA C:2012 8.7 (Advisory)
Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
Variable 'Compute_M' should have internal linkage.

```
#define Compute_IN_Accel                ((uint8_T)1U)
#define Compute_IN_CRUISE               ((uint8_T)1U)
#define Compute_IN_Coast                ((uint8_T)2U)
#define Compute_IN_NO_ACTIVE_CHILD      ((uint8_T)0U)
#define Compute_IN_OFF                  ((uint8_T)2U)
#define Compute_IN_ON                   ((uint8_T)1U)
#define Compute_IN_STANDBY              ((uint8_T)2U)
#define Compute_IN_Steady               ((uint8_T)3U)

/* Block states (auto storage) */
DW_Compute_T Compute_DW;

/* Real-time model */
RT_MODEL_Compute_T Compute_M_;
RT_MODEL_Compute_T *const Compute_M = &Compute_M_;

/* Exported data definition */

/* Definition for custom storage class: Global */
boolean_T AccelResSw;
boolean_T Brake;
boolean_T CoastSetSw;
boolean_T CruiseOnOff;
uint8_T Speed;
boolean_T engaged;
uint8_T tspeed;

/* Definition for custom storage class: Global */
uint8_T holdrate = 5U;
uint8_T incdec = 1U;
uint8_T maxtspeed = 90U;
```

2. In your model, right-click Compute target speed and select **Polyspace > Options**.
3. Set the **Settings from** (Polyspace Bug Finder) option to `Project configuration`. This option allows you to choose a subset of MISRA rules in the Polyspace configuration.
4. Click the **Configure** button.
5. In the Polyspace Configuration window, on the **Coding Rules & Code Metrics** pane, select the check box **Check MISRA C:2012** and from the drop-down list, select

`single-unit-rules`. Now, Polyspace checks only the MISRA C:2012 rules that are applicable to a single unit.



Coding Rules & Code Metrics

Coding Rules

- [ ] Check MISRA C:2004 — required-rules
- [ ] Check MISRA AC AGC — OBL-rules
- [✓] Check MISRA C:2012 — single-unit-rules
- [ ] Use generated code requirements
- [ ] Check custom rules

Effective boolean types: boolean_T

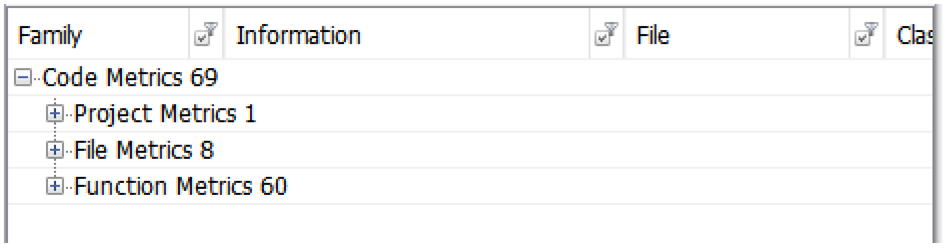
Code Metrics

- [✓] Calculate Code Metrics

6 Save and close the Polyspace configuration window.

7 Rerun the analysis with the new configuration.

When the Polyspace environment reopens, there are no MISRA results, only code metric results. The rules Polyspace showed previously were found because the model was analyzed by itself. When you limited the rules Polyspace checked to the single-unit subset, no violations were found.



Family	Information	File	Clas
⊟ Code Metrics 69			
⊞ Project Metrics 1			
⊞ File Metrics 8			
⊞ Function Metrics 60			

When this model is integrated with its parent model, you can add the rest of the MISRA C:2012 rules.

Generate Report

To demonstrate compliance with MISRA C:2012 and report on your generated code metrics, you must export your results. This section shows you how to generate a report after the analysis. If you want to generate a report every time you run an analysis, see `Generate report`.

1. If they are not open already, open your results in the Polyspace environment.
2. From the toolbar, select **Reporting** > **Run Report**.
3. Select **BugFinderSummary** as your report type.
4. Click **Run Report**.

   The report is saved in the same folder as your results.
5. To open the report, select **Reporting** > **Open Report**.

See Also

Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Bug Finder)
- “Test Two Simulations for Equivalence”
- “Export Test Results and Generate Reports” on page 8-9