Simulink® Test™

User's Guide

MATLAB®&SIMULINK®

R2017a

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Simulink® Test™ User's Guide

Trademarks

Patents

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)

Contents

Test Strategies

1

Test Harness

2

3 Test Sequences and Assessments

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

5 Simulink Test Manager Introduction

6 Test Manager Test Cases

7 Test Manager Results and Reports

8 Real-Time Testing

9 Verification and Validation

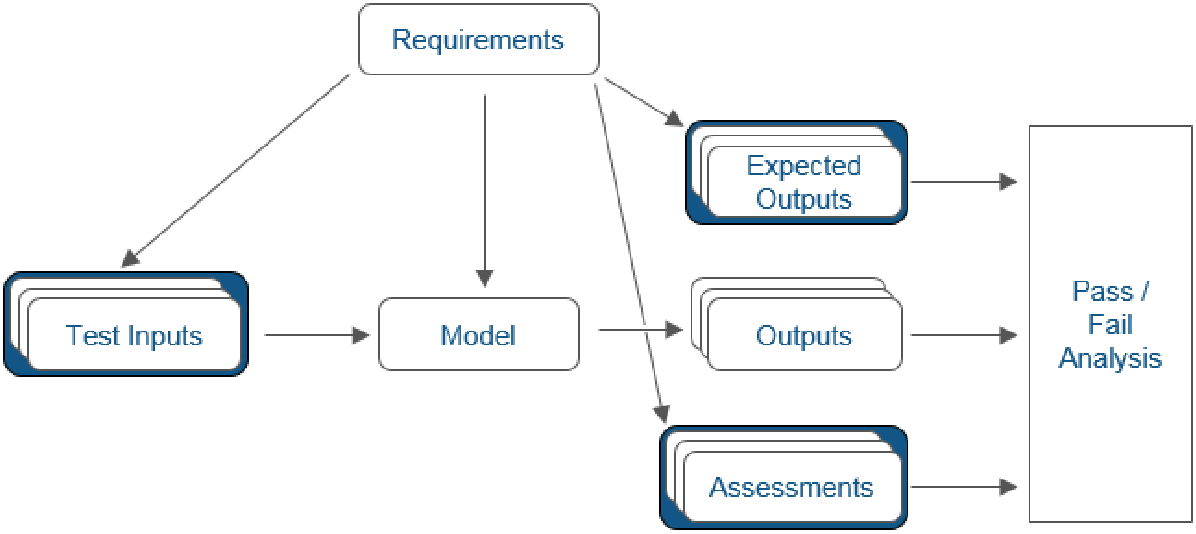
1

Test Strategies

Link Tests 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 Verification and Validation™ license, you can link requirements to test harnesses, test sequences, and test cases. Before adding links, review “Supported Requirements Document Types” (Simulink Verification and Validation) and “Requirements Traceability” (Simulink Verification and Validation).

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 Traceability** > **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 Traceability**, 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 Traceability** > **Link to Current Test Case**.

Requirements Traceability Example

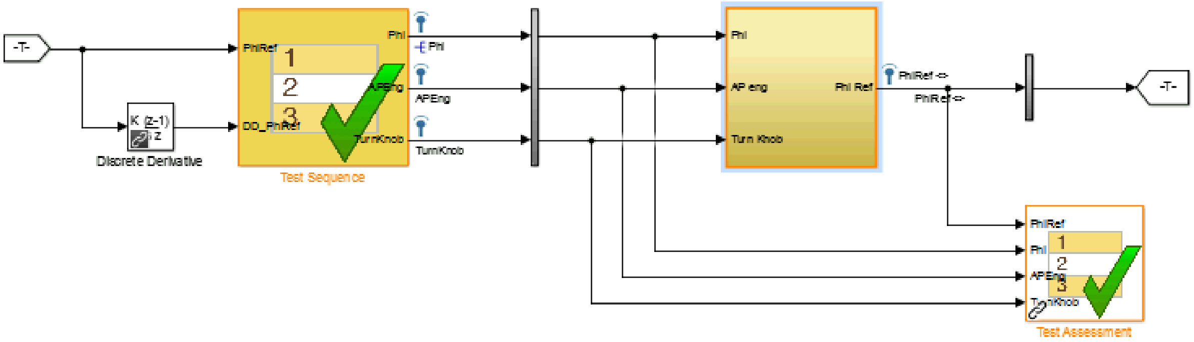
This example demonstrates adding requirements links to a test harness and test sequence. The model is a component of an autopilot roll control system. This example requires Simulink Test and Simulink Verification and Validation.

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 Traceability** > **Highlight Model**.

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



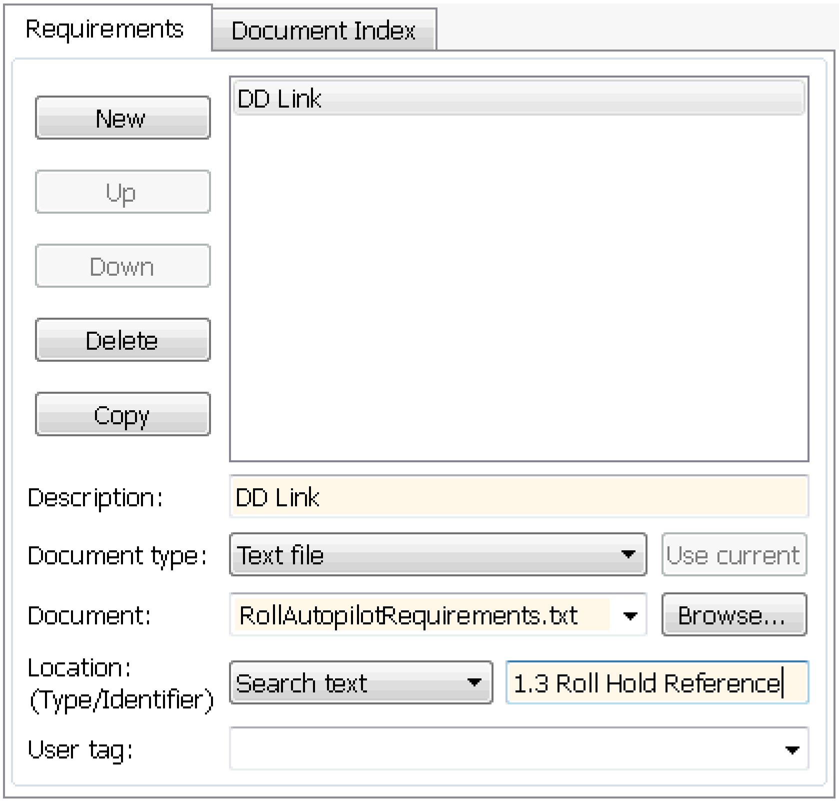
3 Add traceability to the Discrete Derivative block.

a Right-click the Discrete Derivative block and select **Requirements Traceability** > **Open Link Editor**.

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

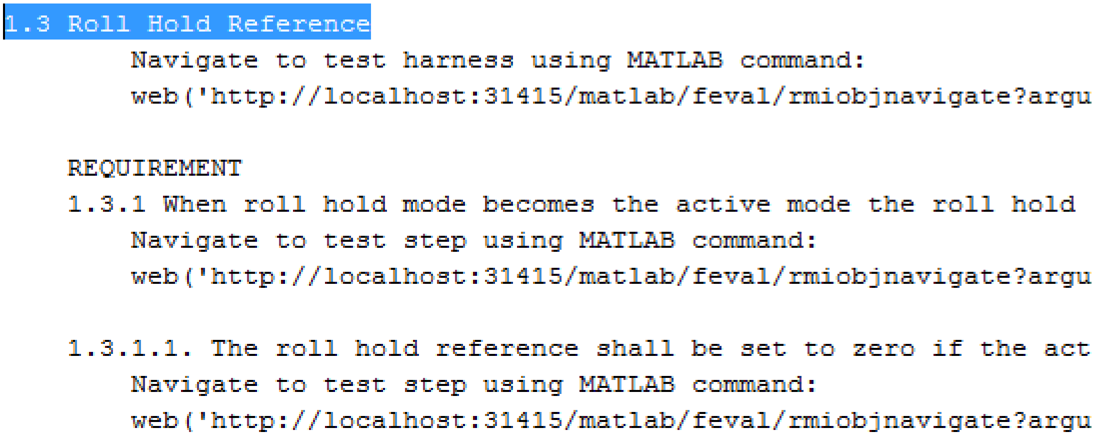
Document: RollAutopilotRequirements.txt

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 Traceability** > **DD Link**. The requirements document opens in the editor and highlights the linked text.



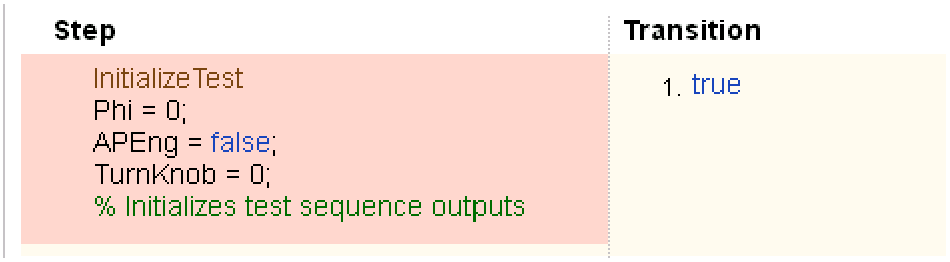
```
1.3 Roll Hold Reference
        Navigate to test harness using MATLAB command:
        web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu

    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 Traceability** > **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

Related Examples

- “Organize Test Sequences” on page 3-8
- “Reuse Test Assessments” on page 3-57
-

Test Harness

Test Harness and Model Relationship

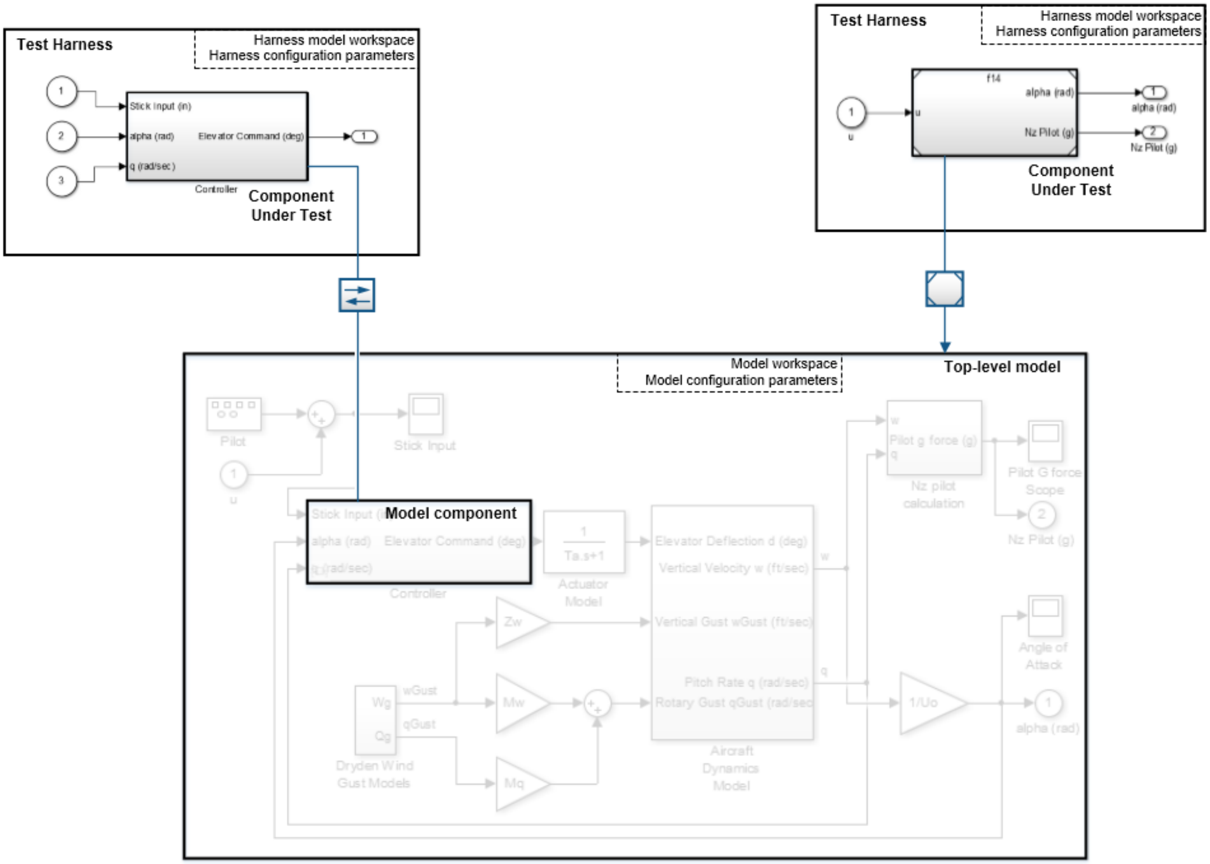
In this section...

Test Harness Description

A test harness is a model block diagram that you can use to develop, refine, or debug a Simulink model or component. 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, yet it persists 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. The test harness isolates the component, providing a separate simulation environment from the main model.
- A top-level model. The component under test is a Model block referencing the main model.

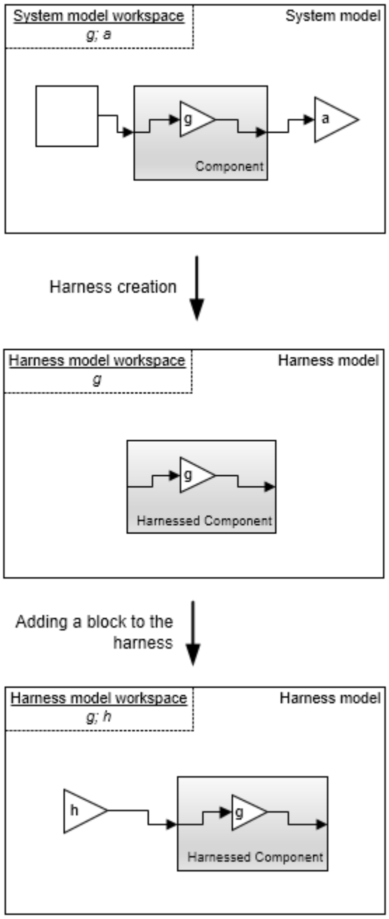


Harness — Model Relationship for a Model Component

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

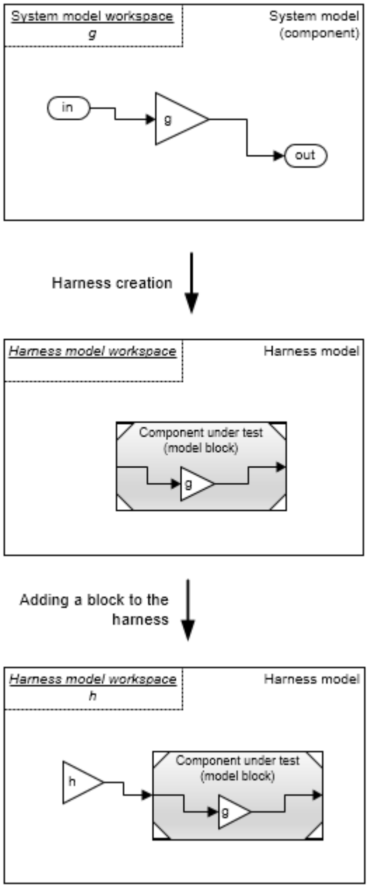
This example shows a test harness for a component that contains a Gain block. The harness model workspace contains a copy of the parameter *g* because *g* defines a part of the component.

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.

More About

- "Componentization Guidelines" (Simulink)

Considerations and Limitations

In this section...
"Test Harness" on page 2-6
"Test Sequence Block" on page 2-6

Consider these behaviors and limitations when working with a test harness or Test Sequence block.

Test Harness

- You can open only one test harness at a time per main model.
- Models in MDL format do not support test harness creation. Convert MDL models to SLX format to use test harnesses. Also, SLX models cannot be saved in MDL format. See "Upgrade Model Files to SLX and Preserve Revision History" (Simulink) in the Simulink documentation.
- 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. Delete all test harnesses before expanding the subsystem.
- Test harnesses are not supported for blocks underneath a Stateflow® object.
- Test harnesses do not support asynchronous sample times.
- 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.

Test Sequence Block

- HDL code generation is not supported for the Test Sequence block.
- The Test Sequence Editor changes the following syntax automatically:

- Duplicate test step names. For example, if `step_1` already 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`.
- The editor inserts explicit casts for literal constant assignments. For example, if `y` is defined as type `single`, then `y=1` is changed to `y=single(1)`.

Select Test Harness Properties

In this section...

Create a Test Harness

To create a test harness for the 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.

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 2-24.

Choosing 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, and you can also use custom sources and sinks from the Simulink Sources or Sinks library. Select `Custom` source or sink, and enter the path to the custom block, such as:

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

```
simulink/Sinks/Terminator
```

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

Use 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. If you use a Test Sequence block source, you can also author assessments directly in the Test Sequence block. See “Reuse Test Assessments” on page 3-57.

Open Harness After Creation

Clear **Open Harness After Creation** to create the test harness without subsequently opening it. This can be useful if you need to create a number of test harnesses in succession.

Create without compiling the model

When you select this property, the main model does not compile when generating the test harness. The test harness does not contain conversion subsystems, configuration parameters, or model workspace data for the component under test. The test harness can require additional modification for it to compile, such as adding signal conversion blocks.

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 2-41.

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 2-41.

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.

- `Synchronize on harness open and close`: The component in the main model, or the test harness, is updated when the harness closes or opens. Select this option if you plan to do repeated testing and design updates.

- `Synchronize on harness open`: The component in the test harness is updated when the harness opens. Select this option if your design is largely complete, or to prevent changes in the test harness.
- `Synchronize only during push and rebuild`: Synchronization does not occur when the harness opens or closes. Select this option to manually control design updates, by selecting **Analysis** > **Test Harness** > **Push Component and Parameters to Main Model** or **Analysis** > **Test Harness** > **Rebuild Harness from Main Model**.

Initialize/Terminate/Reset Behavior

Generate scheduler for Initialize, Reset, and Terminate tasks

Testing a model with initialize, terminate, or reset behavior can require calling Initialize, Terminate, or Reset subsystems to set the desired state. You can use the Test Sequence block to schedule function calls using the `send()` command and function-call outputs.

You can automatically create a Test Sequence block configured to schedule function calls to Initialize, Terminate, or Reset inputs. When you create a test harness, select **Generate scheduler for Initialize, Reset, and Terminate tasks** in the **Advanced Properties** tab of the create harness dialog box. After test harness creation, the Test Sequence block contains template function calls for use in your test sequence.

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**.

Change Harness Properties

Click the badge ⊡ in the test harness block diagram and click **Test harness properties** to open the harness properties dialog box.

See Also

Test Sequence | “Synchronize Changes Between Test Harness and Model” on page 2-41

Test Harness Parameters and Signals

In this section...
"Test Harness Generation Without Compilation" on page 2-13
"Signal Conversion Subsystem" on page 2-13

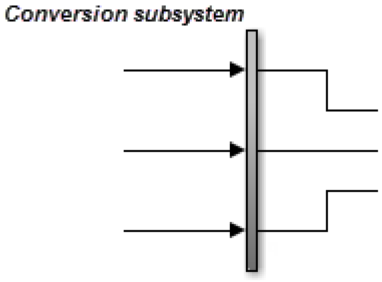
Test Harness Generation Without Compilation

You can generate a test harness without compiling the main model. For example, this option can be useful if you are prototyping a design that cannot yet compile. If the main model is not compiled when generating a test harness:

- 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.

To execute these processes, 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 2-41.

Signal Conversion Subsystem



A signal conversion subsystem

- Contains signal specification blocks to check signal properties to and from the component under test. These properties include data type, sample time, bus properties, dimension, and complexity.
- Contains blocks that simplify signal routing in the test harness block diagram, such as Goto and Function-Call Split blocks.

Signal types must match the signal specification for test harnesses to compile. If you get a compile error related to the signal conversion subsystem, check the signal properties and consider modifying the test harness design. For example:

- Add conversion blocks to your test harness outside the conversion subsystem.
- Edit the conversion subsystem. The subsystem is locked by default. To unlock it, right-click the subsystem, select **Block Parameters**, then set **Read/Write permissions** to `ReadWrite`.

Note: When you rebuild the test harness, the signal conversion subsystems are rebuilt. If you modify a conversion subsystem, disable automatic test harness rebuild to avoid losing your modifications when you open the test harness. See “Select Test Harness Properties” on page 2-8.

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

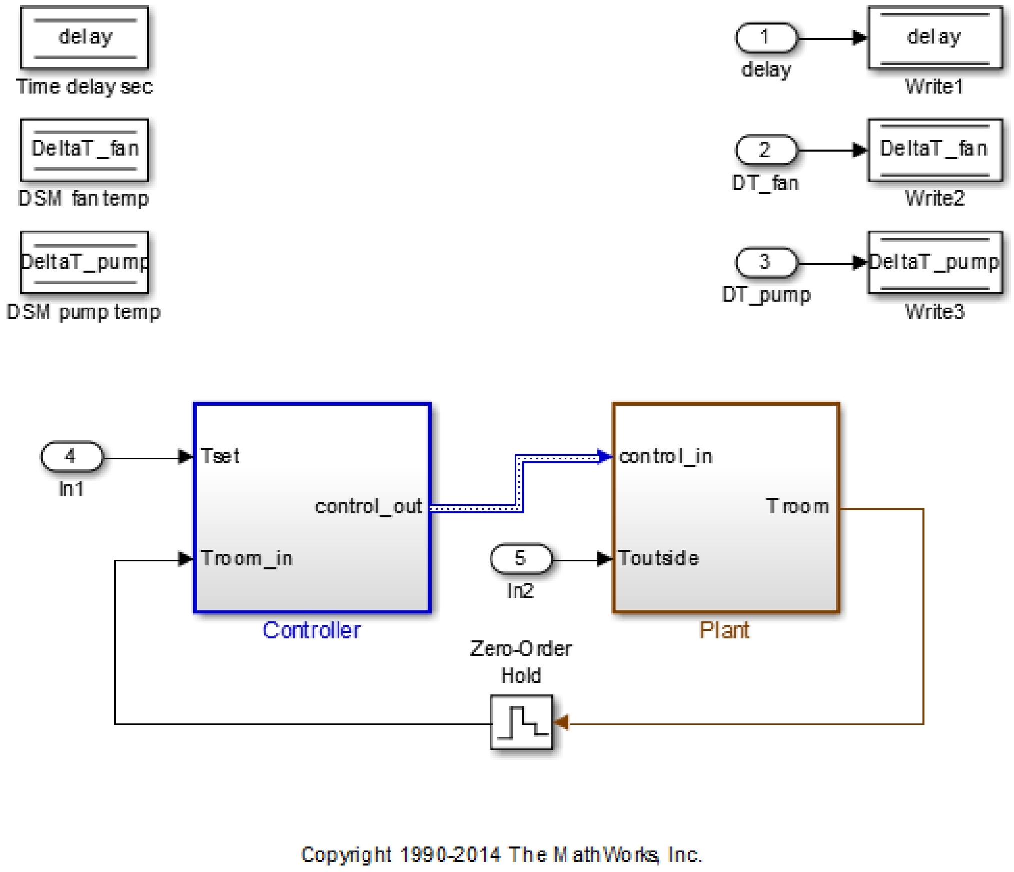
```
cd(fullfile(docroot,'toolbox','sltest','examples'))
```

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

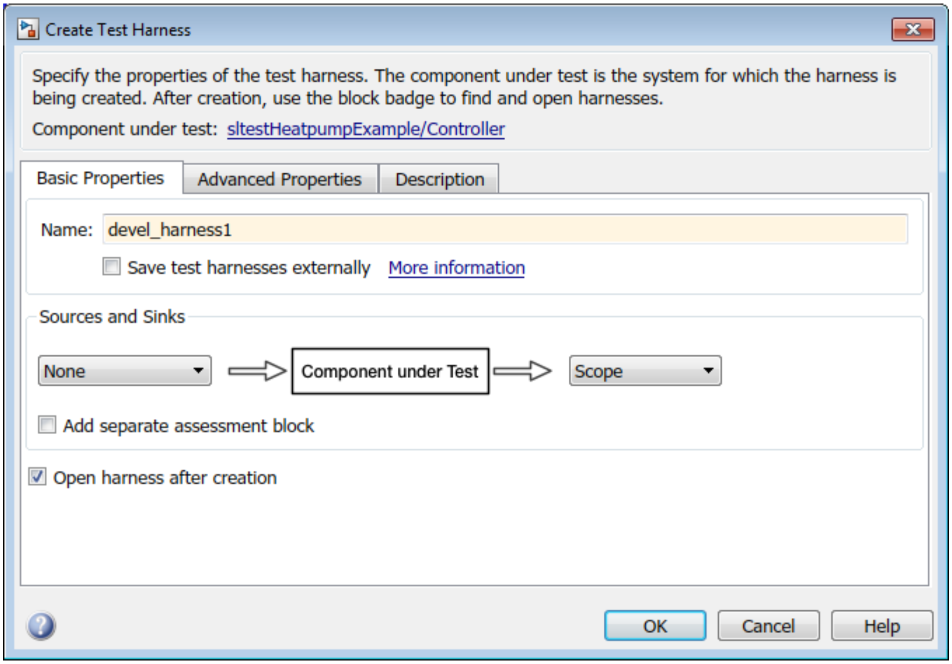
1. Right-click the `Controller` subsystem and select **Test Harness > Create for ‘Controller’**.
2. Set the harness properties:

   In the **Basic Properties** tab:

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

   In the **Advanced Properties** tab:

   - **Initial harness configuration**: `Refinement/Debugging`



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

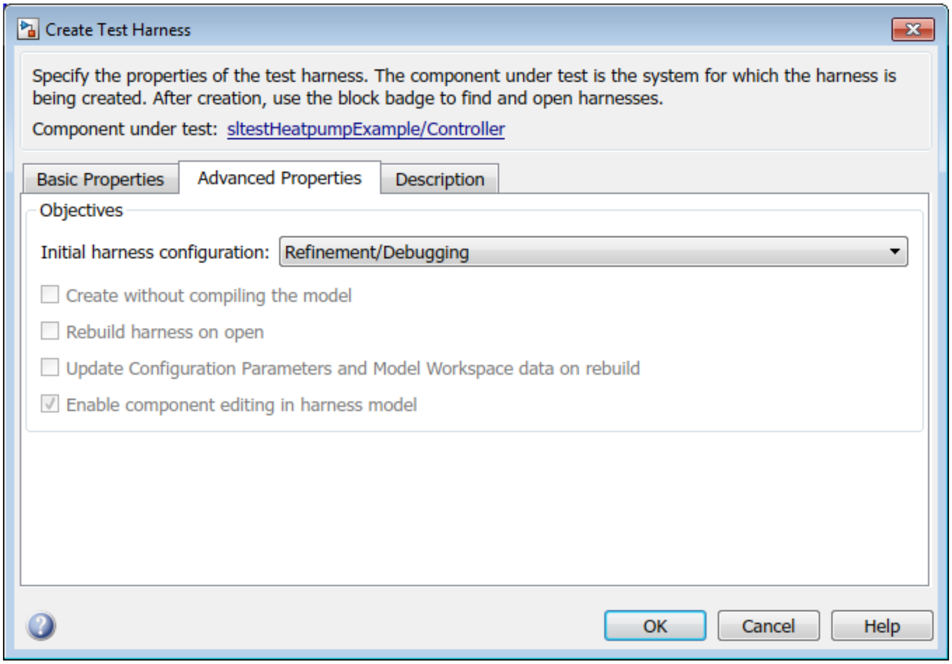
□ Save test harnesses externally

Sources and Sinks

None → Component under Test → Scope

□ Add separate assessment block

☑ Open harness after creation



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

Objectives

Initial harness configuration: Refinement/Debugging

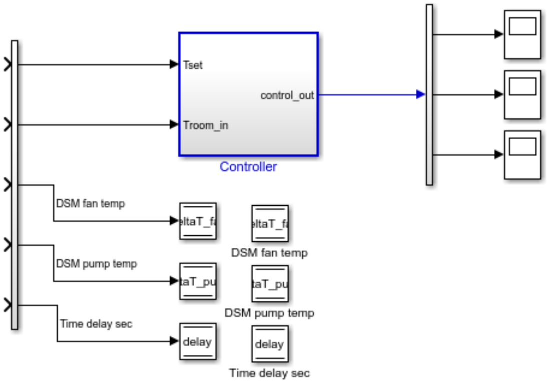
□ Create without compiling the model

□ Rebuild harness on open

□ Update Configuration Parameters and Model Workspace data on rebuild

☑ Enable component editing in harness model

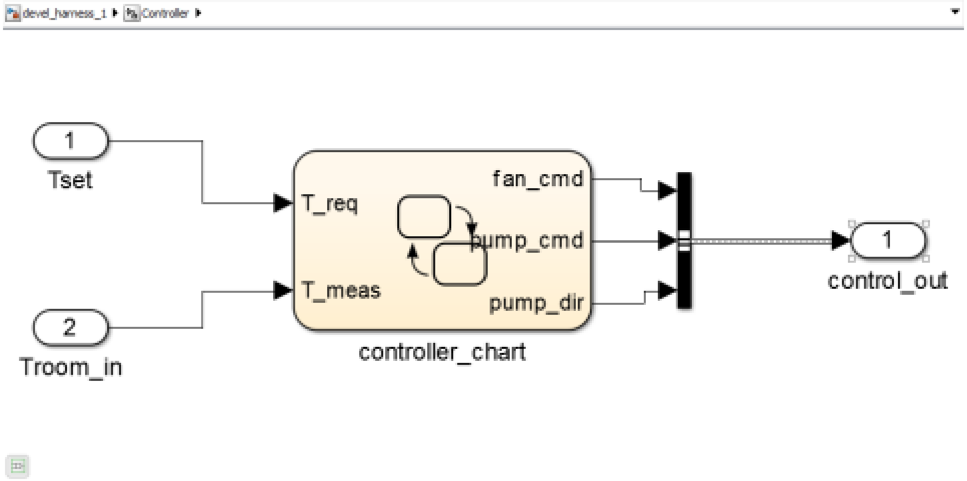
3 Click **OK** to create the test harness.



Inspect and Refine the Controller

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

2 Connect the chart to the inports.



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

Add a Test Case 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`.

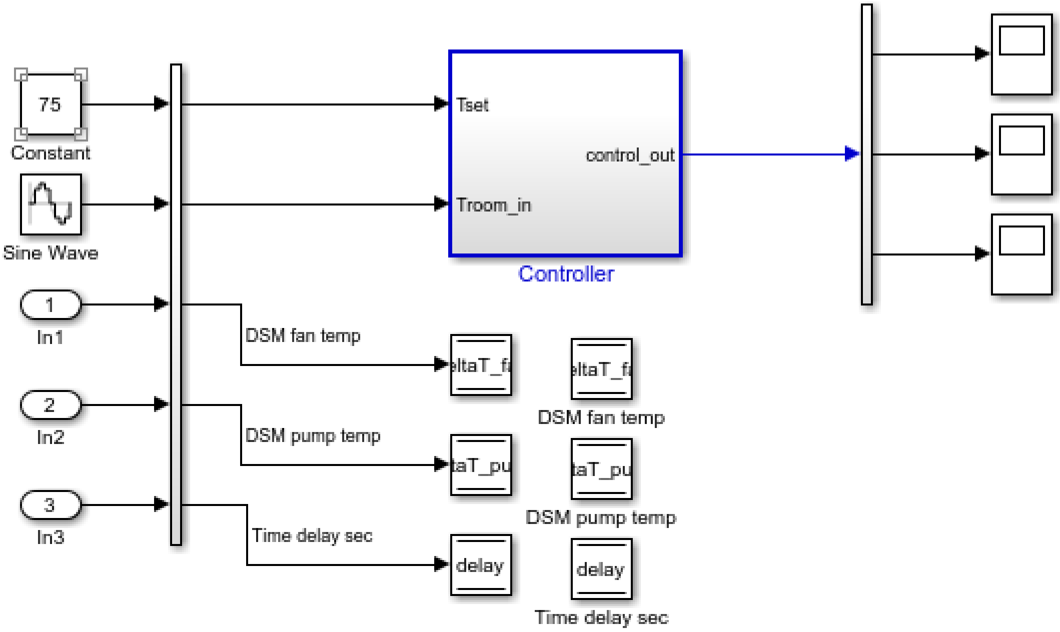
3 Add a Sine Wave block to the harness model to simulate a temperature signal. Connect the Sine Wave block to the conversion subsystem input `Troom_in`.

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`.

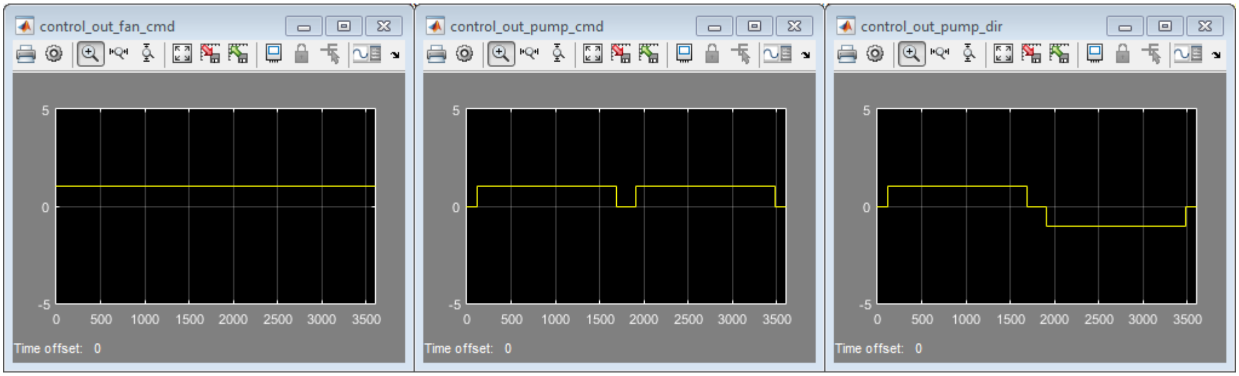
8 Open the three scopes in the harness model.

9 Simulate the harness.

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`.



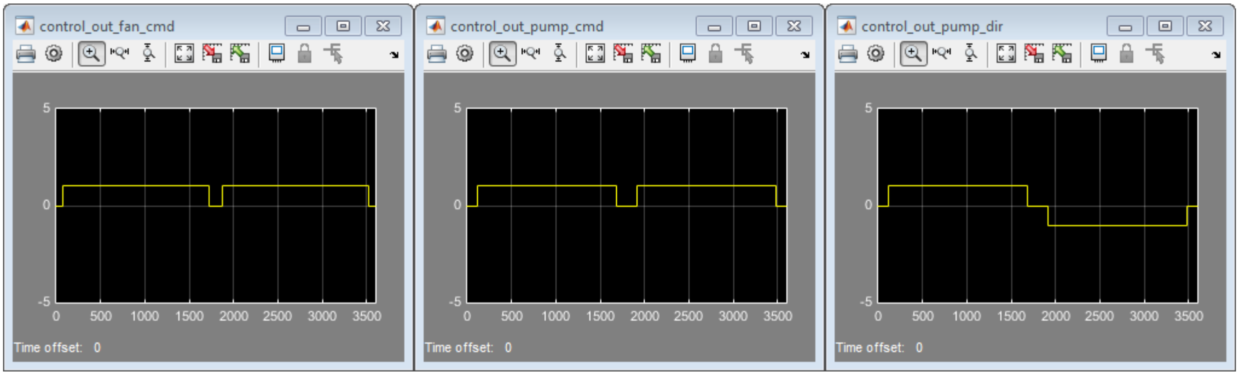
2 In the harness model, open the `Controller` subsystem.

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`.

Related Examples

- "Test a Model Component Using Signal Functions" on page 3-48
- "Test Downshift Points of a Transmission Controller" on page 3-51

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 “Select Test Harness Properties” on page 2-8.

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.

- 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

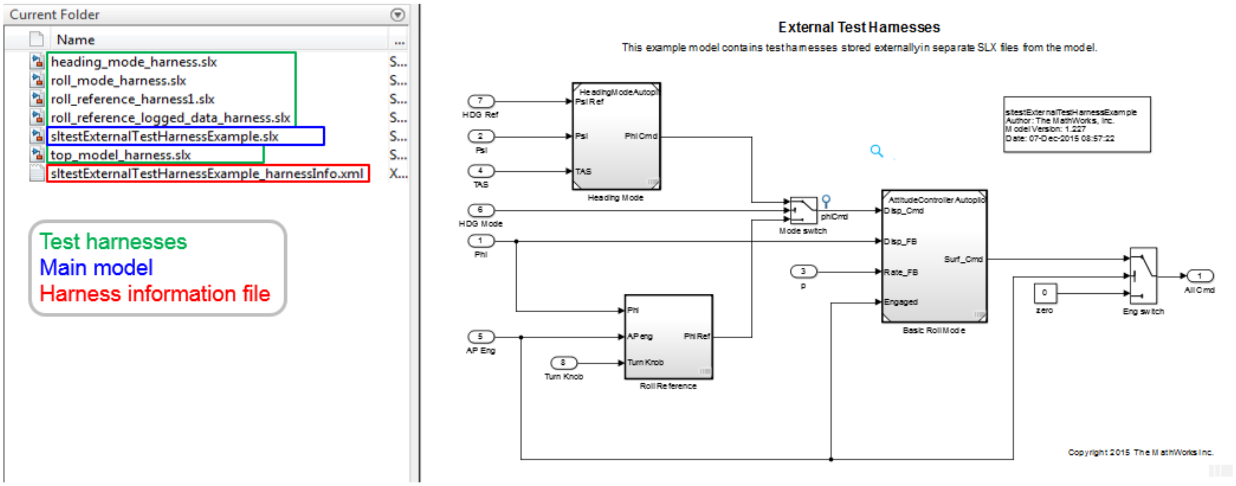
You can change how your model stores test harnesses at different phases of your model lifecycle. For example:

- 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.

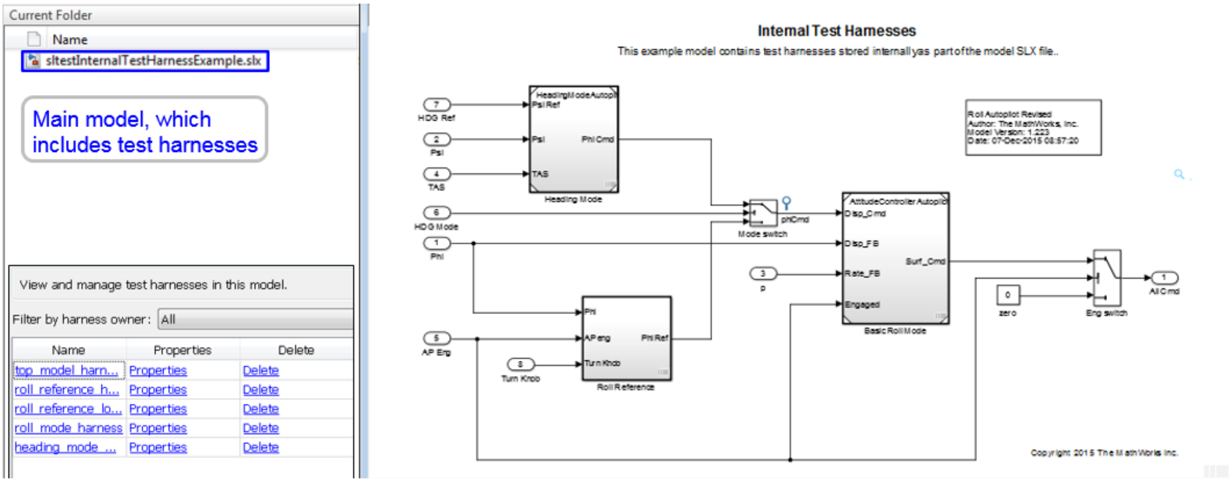
   The harnesses are converted.
4. The conversion to external test harnesses creates an SLX file for each test harness and a harness information XML file `<modelName>_harnessInfo.xml`.



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...

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

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

Internal Test Harnesses

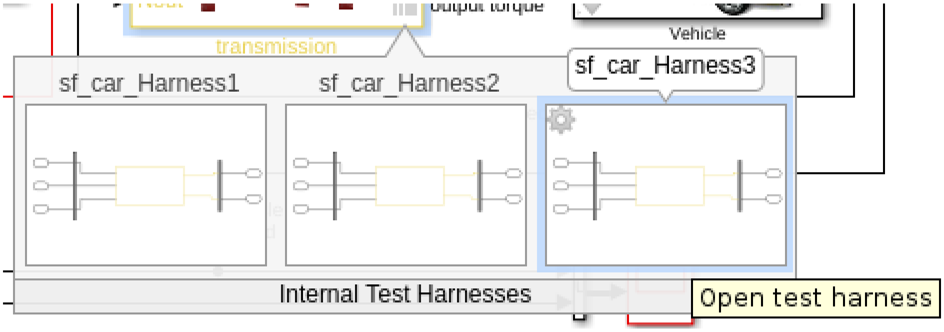
This example model contains test harnesses stored internallyas partofthe model SLX file..

Roll Autopilot Revised
Author: The MathWorks, Inc.
Model Version: 1.223
Date: 07-Dec-2015 08:57:20

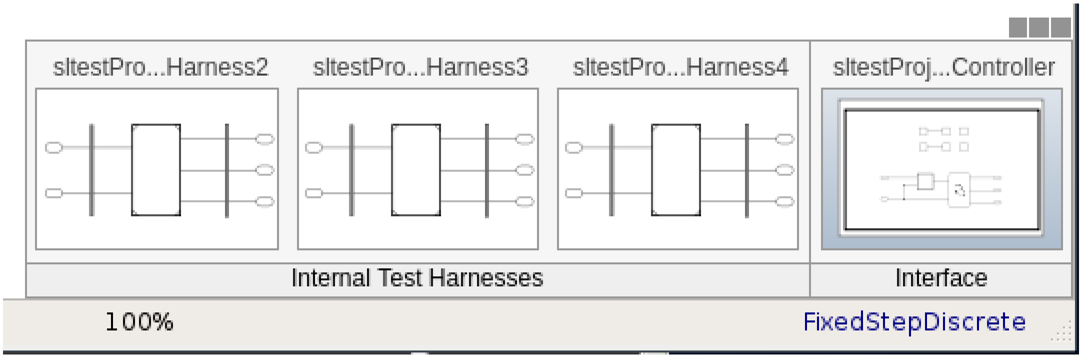
Copyright 2015 The MathWorks Inc.

Preview and Open a Test Harness

When a model component has a test harness, a badge appears in the lower right of the block. Click the badge to preview test harnesses, and click a thumbnail image to open the harness.

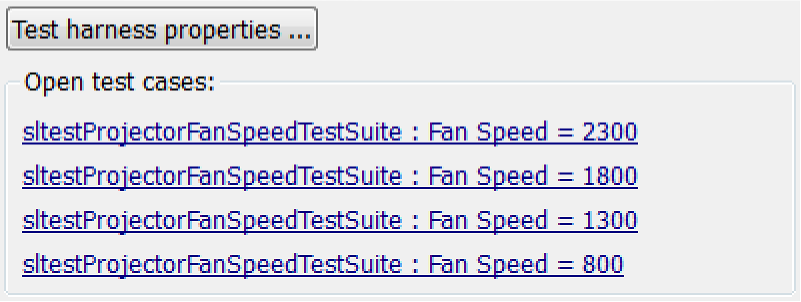


When a model block diagram has a test harness, click the pullout icon in the model canvas to preview the test harnesses. To open the harness, click a thumbnail.



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.



Test harness properties ...

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.

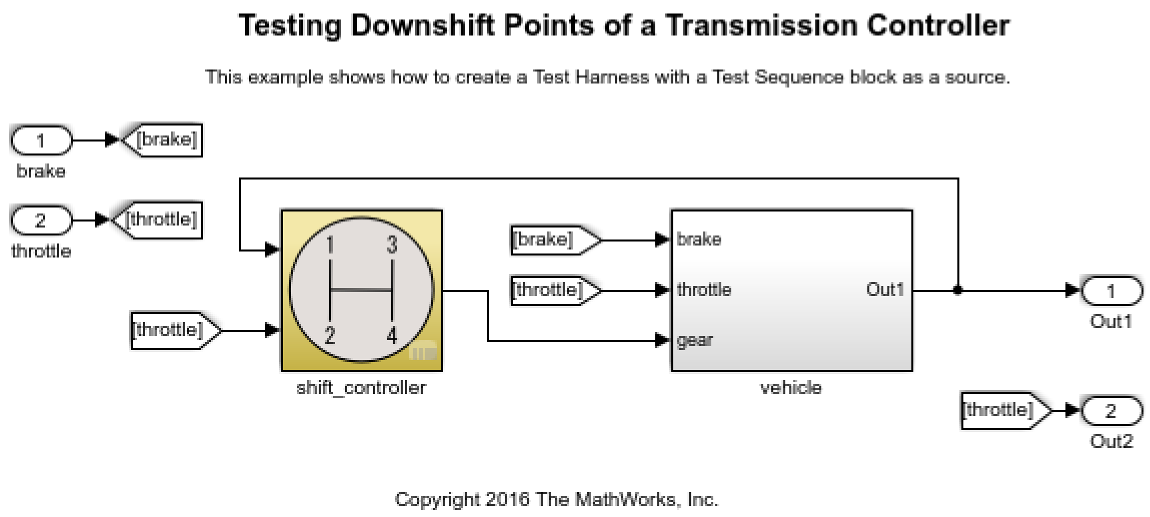
High-level Workflow

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.

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: 13.0028
        ownerFullPath: 'sltestTestSequenceExample/shift_controller'
            ownerType: 'Simulink.SubSystem'
               isOpen: 0
          canBeOpened: 1
             lockMode: 0
     verificationMode: 0
```

```
          saveExternally: 0
           rebuildOnOpen: 0
        rebuildModelData: 0
               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`.

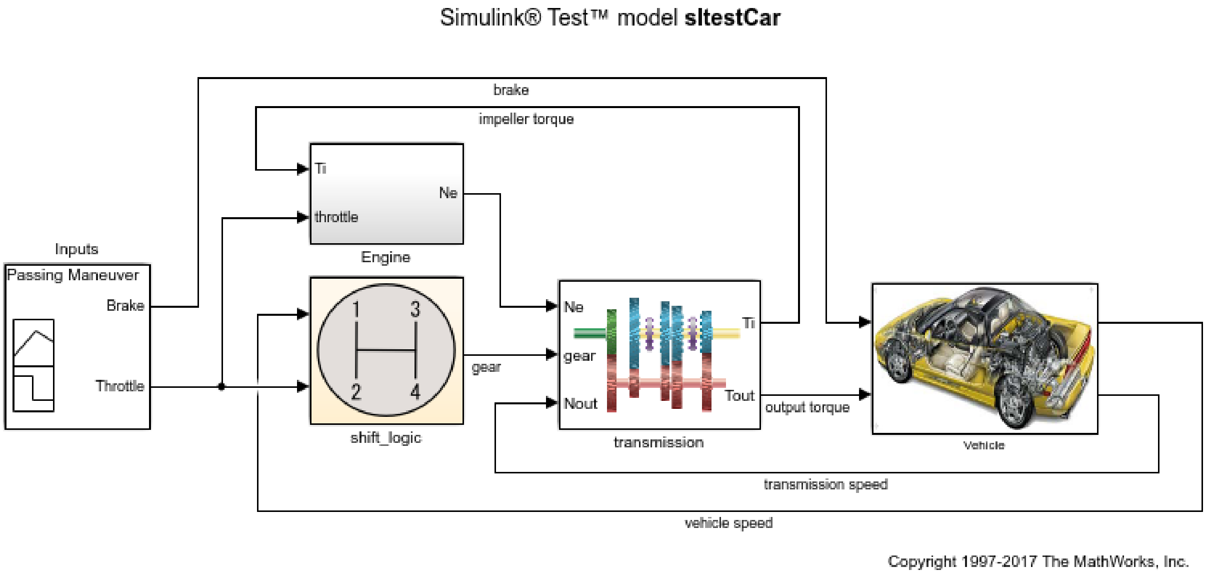
```
sltest.harness.export([model '/shift_controller'],'ControllerHarness2',...
    'Name','ControllerTestModel')

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 = 

  1×4 struct array with fields:

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

4. Delete the harnesses.

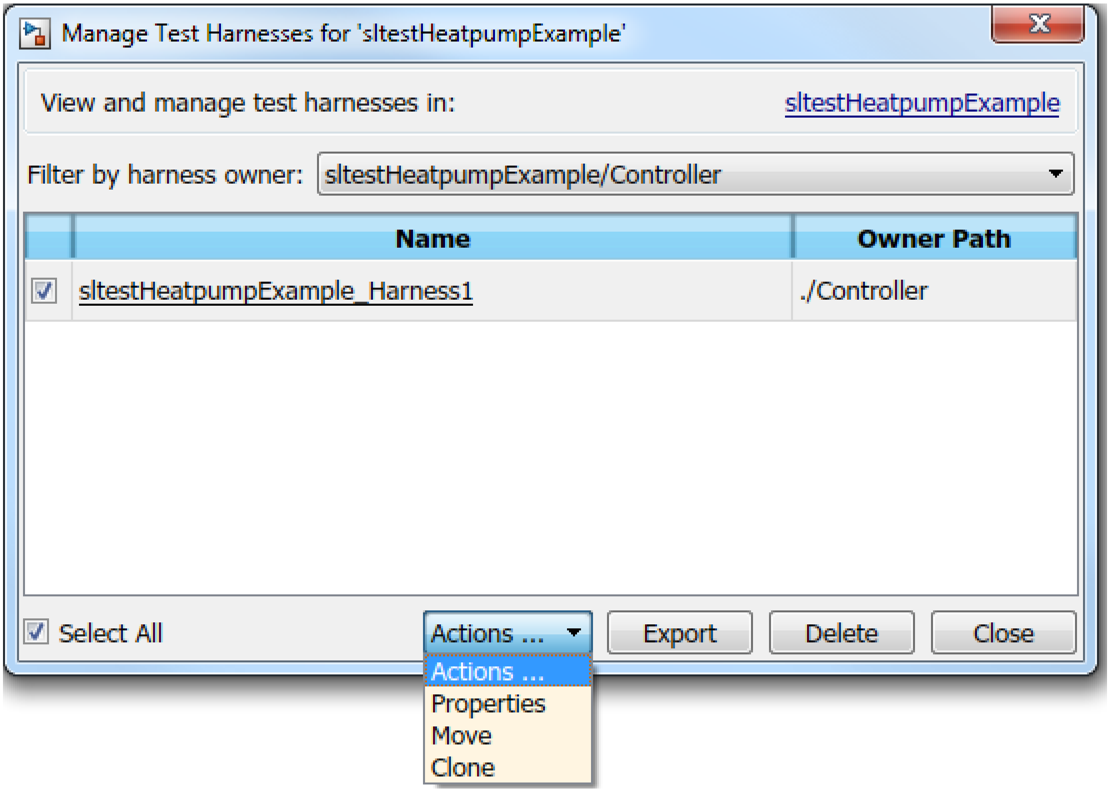
```
for k = 1:length(test_harness_list)
     sltest.harness.delete(test_harness_list(k).ownerFullPath,...
     test_harness_list(k).name)
end

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 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.



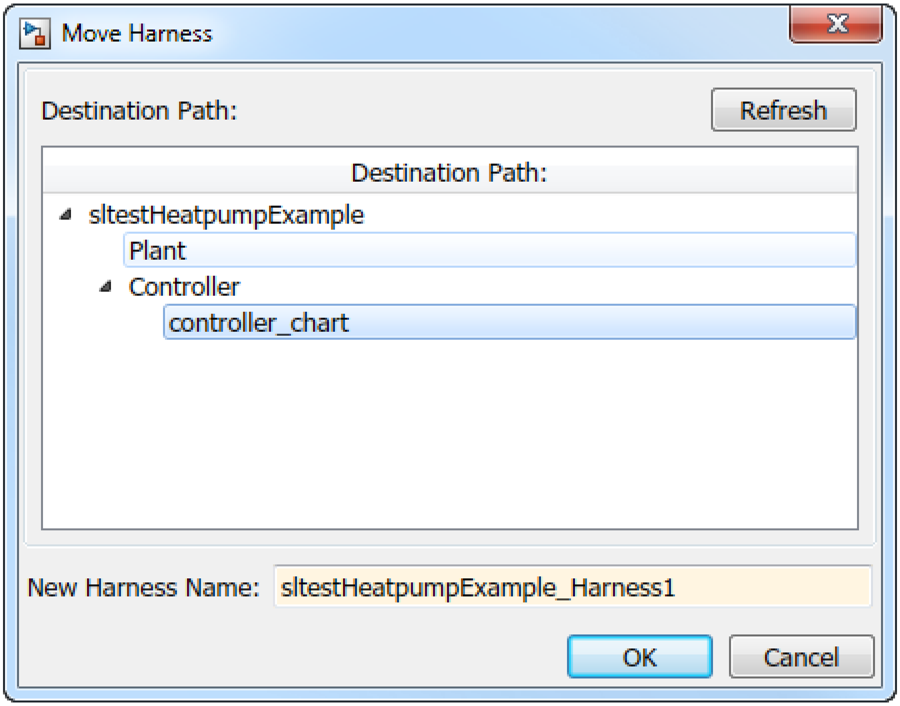
Manage Test Harnesses for 'sltestHeatpumpExample'

View and manage test harnesses in: sltestHeatpumpExample

Filter by harness owner: sltestHeatpumpExample/Controller

	Name	Owner Path
☑	sltestHeatpumpExample_Harness1	./Controller

☑ Select All



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`

Create Test Harnesses from Standalone Models

In this section...

For testing, a common model architecture uses a Signal Builder block to pass test inputs to a copy of a subsystem or a Model block referencing your main model. This architecture is an example of a standalone model designed for testing. Standalone test models include models created by Simulink Verification and Validation, Simulink Design Verifier™, or any custom models.

You can simplify testing by importing standalone models to your main model as Simulink Test test harnesses. Importing standalone models as test harnesses enables synchronization and management features, allowing you to

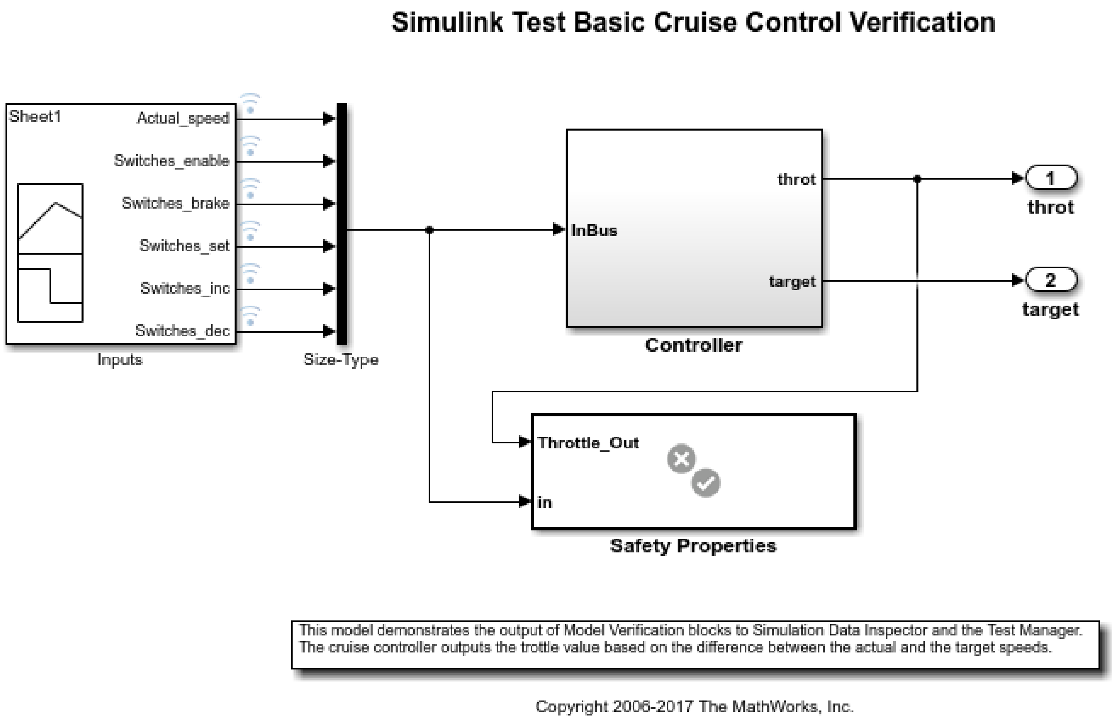
- Iterate on your design, using model – harness synchronization to update your design.
- Manage test harnesses, using the UI and programmatic interface.
- Establish clear ownership of a test harness by a model, subsystem, or library being tested.

Test Harness Import Workflow

Before importing a model as a test harness, you need to know:

- The model or component to associate the test harness with, which can be a top-level model, a subsystem, or a linked block.
- The path to the model to be imported.
- In the model to be imported, the component that is tested. In the new test harness, this component links to the owner component in the main model. You can synchronize design changes when the test harness is open or closed.

  For example, this model uses a Signal Builder block to pass test inputs to the `Controller` subsystem, and a subsystem to verify the `Controller` output. The `Controller` subsystem is tested.



Simulink Test Basic Cruise Control Verification

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.

Harness Import Considerations

You cannot create a test harness by importing

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

When importing, consider the block type in the main model and the standalone model:

- For a user-defined function block in the main model (such as an S-Function block), the tested component in the standalone model must be the same block type.

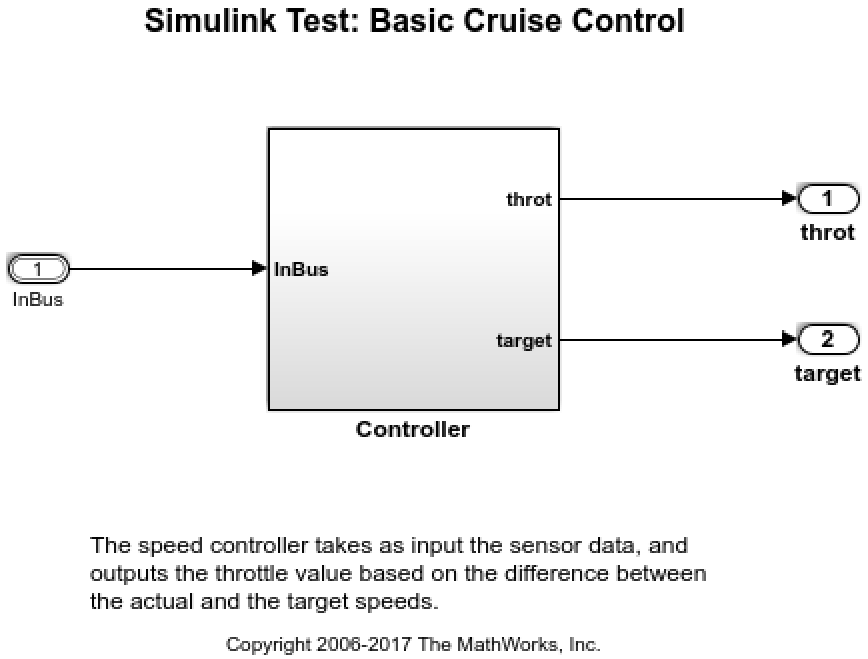
- For a top-level main model, the tested component in the standalone model can be a Model block or a subsystem.
- For a subsystem in the main model, the tested component in the standalone model can be a subsystem, Model block, or any user-defined function block.
- For a Model block in the main model, the tested component in the standalone model can be a Model block or a subsystem.

Import a Standalone Model as a Simulink® Test™ Harness

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

The Main Model and the Harness Model

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

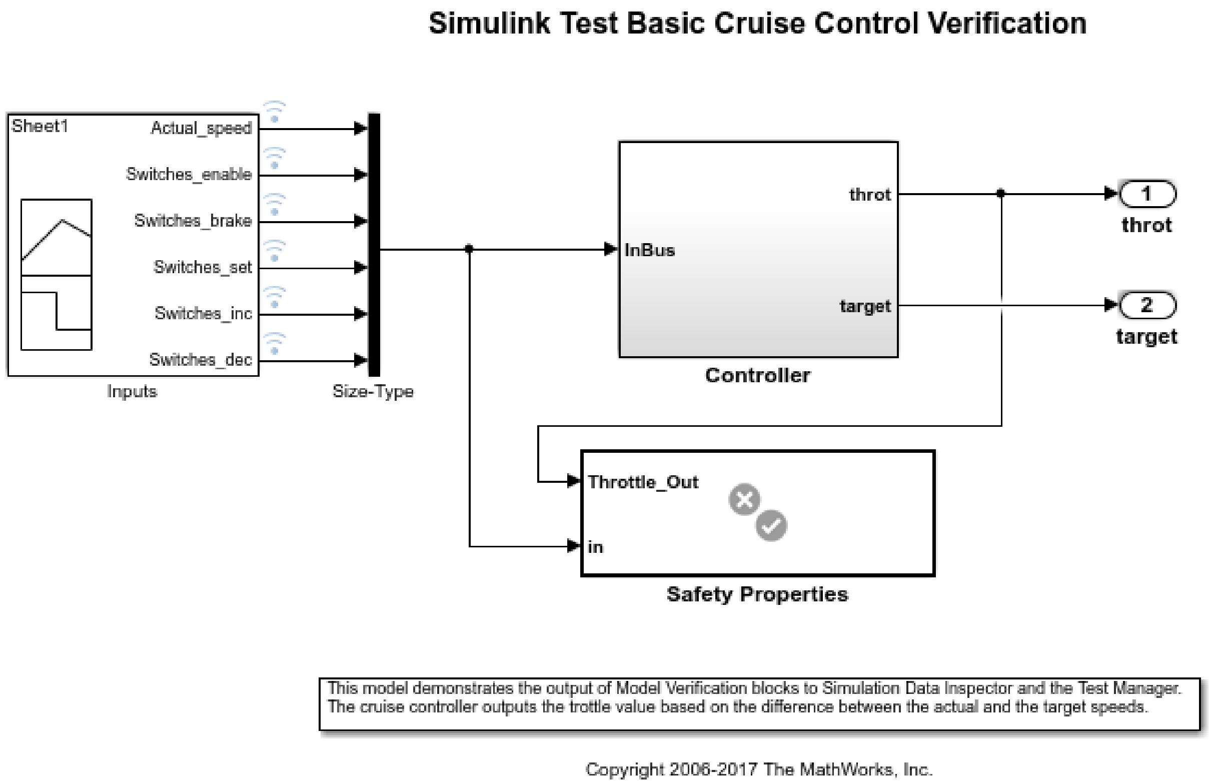


Simulink Test: Basic Cruise Control

The speed controller takes as input the sensor data, and outputs the throttle value based on the difference between the actual and the target speeds.

Copyright 2006-2017 The MathWorks, Inc.

The standalone 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.



Simulink Test Basic Cruise Control Verification

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.

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'**.

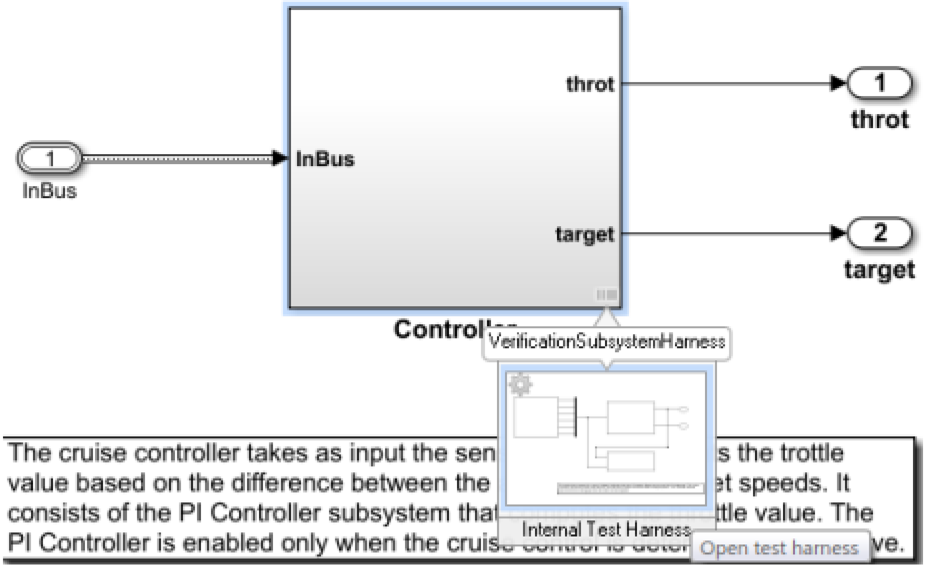
2. Set the following harness properties:

- **Name**: `VerificationSubsystemHarness`
- Clear **Save test harness externally**

- **Simulink model to import:** Click **Browse** and select `sltestCruiseControlHarnessModel`.
- **Component under Test in imported model**: `Controller`

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.



The cruise controller takes as input the sen s the trottle
value based on the difference between the et speeds. It
consists of the PI Controller subsystem tha ttle value. The
PI Controller is enabled only when the cruis ve.

See Also

`sltest.harness.import`

Synchronize Changes Between Test Harness and Model

In this section...

A test harness lets you synchronize changes between the test harness and the main model. You can transfer a configuration set and model workspace variables, update the component design, and rebuild the harness to reflect the latest model design.

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, regularly rebuild your test harness so that the generated code referenced by the SIL/PIL block reflects the current main model. You can set a test harness to rebuild every time it opens. Open the test harness properties dialog box by clicking the test harness badge in the harness model and select **Rebuild harness on open**.

To minimize compilation, you can manually rebuild the test harness if you have a large or complex main model. You can check the SIL/PIL block equivalence to determine whether to rebuild the harness. In the harness model, from the menu bar, select **Analysis > Test Harness > Compare Checksums**, which compares the checksum of the component in the model to the checksum archived during the SIL/PIL block generation. If the result is different, rebuild the harness by clicking **Analysis > Test Harness > Rebuild Harness from Main Model**.

For information about running multiple simulations with unchanged generated code, see "Prevent Code Changes in Multiple Simulations" (Embedded Coder).

Synchronize Changes to the Component Under Test

CUT Synchronization Options

You can specify when the component under test is synchronized with the main model. Synchronization is controlled by `SynchronizationMode` property.

- `Synchronize on harness open and close`: The component in the main model, or the test harness, is updated when the harness closes or opens. Select this option if you plan to do repeated testing and design updates.
- `Synchronize on harness open`: The component in the test harness is updated when the harness opens. Select this option if your design is largely complete, or to prevent changes in the test harness.
- `Synchronize only during push and rebuild`: Synchronization does not occur when the harness opens or closes. Select this option to manually control design updates, by selecting **Analysis** > **Test Harness** > **Push Component and Parameters to Main Model** or **Analysis** > **Test Harness** > **Rebuild Harness from Main Model**.

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.

Change the Synchronization Mode

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 block diagram 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.

Rebuild Test Harness

You can 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**. This operation rebuilds conversion subsystems in the test harness. If the test harness does not have conversion subsystems, this process adds them.

Depending on your test harness settings, harness rebuild can also copy parameters and the active model configuration set. For example, suppose that you update the component design to use a new parameter. When you rebuild the harness, the harness model workspace receives a copy of the parameter.

To copy parameters and the model configuration set, when you create or modify the properties of a test harness, select **Update Configuration Parameters and Model Workspace data on rebuild**.

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.

Also see “Select Test Harness Properties” on page 2-8 and `sltest.harness.rebuild`.

Update Parameters from Test Harness to Model

When working in the test harness, you can add a workspace item to the harness model workspace or change the test harness configuration set. To update the configuration set and workspace in the main model, select **Analysis > Test Harness > Push Parameters to Main Model**. This operation:

- Copies the active configuration set from the harness model to the main model, and makes it the active configuration set in the main model.
- Copies workspace contents to the main model, if the contents are relevant to the component under test.

This example shows how to push a new workspace variable to the main model.

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:

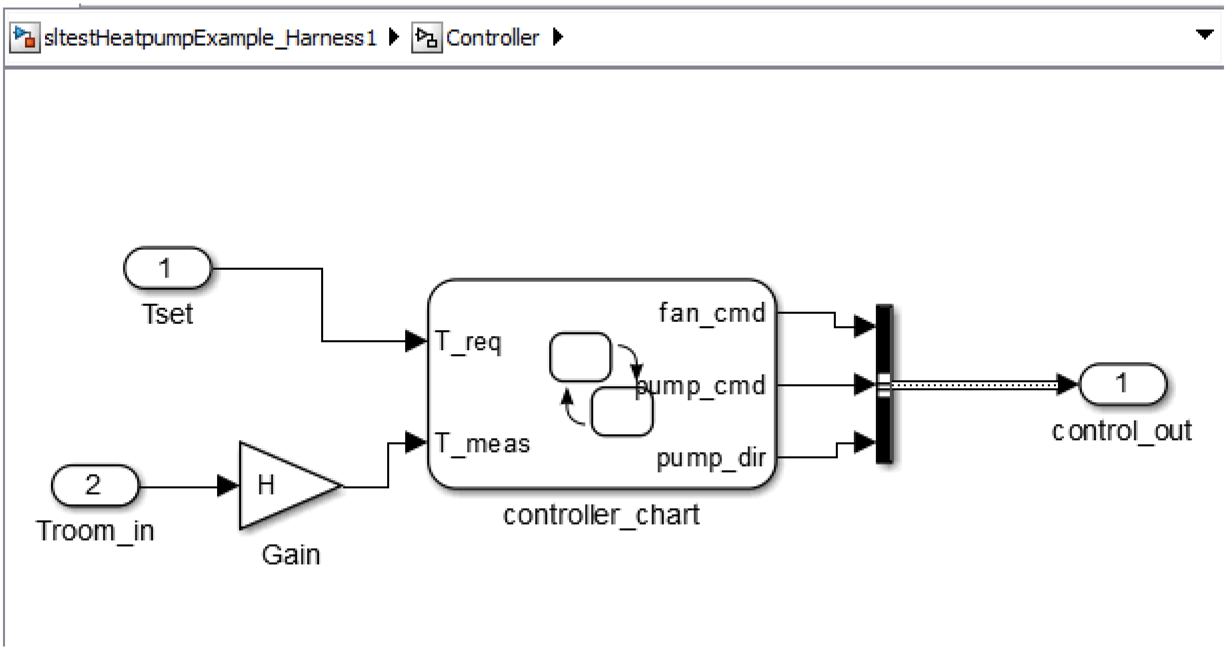
   ```
   sltestHeatpumpExample.slx
   sltestHeatpumpBusPostLoadFcn.mat
   PumpDirection.m
   ```
3. Open the model.

   ```
   open_system('sltestHeatpumpExample')
   ```
4. Right-click the `Controller` subsystem and select **Test Harness > Create Test Harness**.
5. In the Create Test Harness dialog box, click **OK** to create a test harness with default properties. The test harness model opens.

6 In the test harness model, select **Tools > Model Explorer** to open the Model Explorer. Expand the items under the test harness name and select **Model Workspace**.

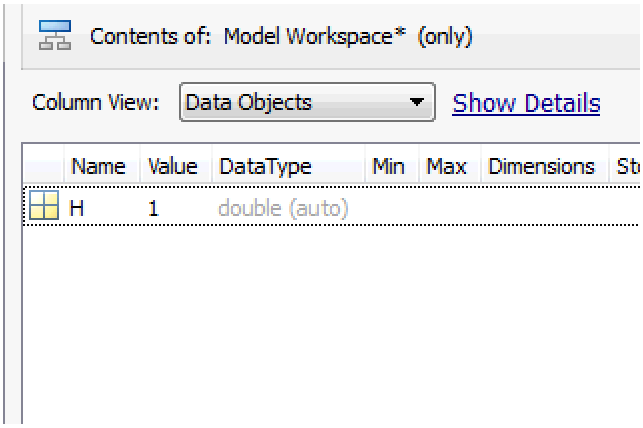
7 Select **Add > MATLAB Variable**. Set the variable name to `H` and the value to `1`.

8 In the top level of the test harness, double-click `Controller` to open the subsystem. Add a Gain block and set the value to `H`. Connect it as shown.



9 Select **Analysis > Test Harness > Push Parameters to Main Model**.

10 In the Model Explorer, expand the main model and select **Model Workspace**. `H` appears as a variable in the workspace.



Contents of: Model Workspace* (only)

Column View: Data Objects

Name	Value	DataType	Min	Max	Dimensions	St
H	1	double (auto)				

Related Examples

- “SIL Verification for a Subsystem” on page 4-2

Test Library Blocks

In this section...

You can use a library subsystem to help facilitate component reuse. Design and test workflows can require testing of a reusable component source and each instance of the component. For libraries, you can set up tests for the library subsystem during your design. Once the library subsystem meets your requirements, you can create linked blocks in larger models and test the subsystem instances.

Library Testing Workflow

Library testing broadly divides between testing the source library subsystem, and testing each instance of the library subsystem. Testing the library subsystem checks the design in isolation, while testing each instance checks the component in the context of the larger system. Test harnesses can move from the source to the instance and the instance to the source.

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. Use this test case to perform requirements-based tests.
2. Test the library subsystem. If it fails your requirements, edit the model and run the test case again.
3. Lock the library after the subsystem meets the requirements.
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 Harness

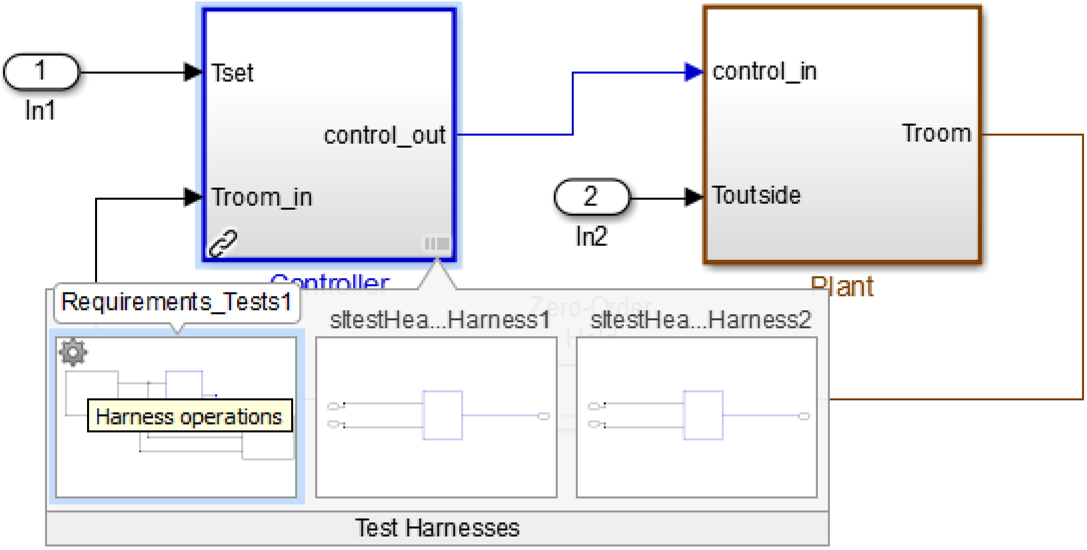
A test harness for a library subsystem has specific properties, compared to test harnesses for a subsystem in a model.

- Libraries do not compile, so a test harness for a library subsystem does not contain compiled attributes.
- 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 2-24.

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. This linked subsystem has three test harnesses. To move the `Requirements_Tests1` test harness,

1. On the linked subsystem, click the harness badge.
2. Click the **Harness Operations** ⚙ icon on the test harness you want to promote.



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 on 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 “Select Test Harness Properties” on page 2-8.

Related Examples

-

Test Sequences and Assessments

Introduction to Test Sequences

In this section...

You can use the Test Sequence block to specify test steps, actions, and transitions. With timeseries inputs, you supply time-defined test vectors. However, the test sequences you create can react to signal and temporal conditions. You can also use them to assess simulation.

Structure of a Test Sequence

A test sequence consists of test steps arranged in a hierarchy. You can use transitions to define the test sequence progression within a hierarchy level.

A test step contains actions and transitions you define using MATLAB as the action language. Actions execute at the beginning of the step. You use actions to define commands for each test step, such as setting signal levels, verifying logical conditions, or setting variables. You use test step transitions to define conditions that determine when the test sequence exits the current step and enters another step.

A standard transition occurs on a condition that you specify. Once the step exits, the next step that you specify executes.

Test Sequence Hierarchy

Arrange the test sequence hierarchy using 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.

Step Transitions

In a test sequence, the top hierarchy level uses a standard transition. Test sequence execution begins with the top step in the group, and proceeds according to the transition conditions and next steps.

You can change lower-level groups to switch between steps based on signal conditions defined in the step. This switching condition is called a When decomposition. In this case, the parent step evaluates, and then the substeps execute based on their associated conditions. The conditions determine the order in which the substeps execute. For example, the first substep in the table does not necessarily execute first. If multiple steps in a When decomposition group have conditions that are true, the highest step with the true condition is active.

Create a Basic Test Sequence

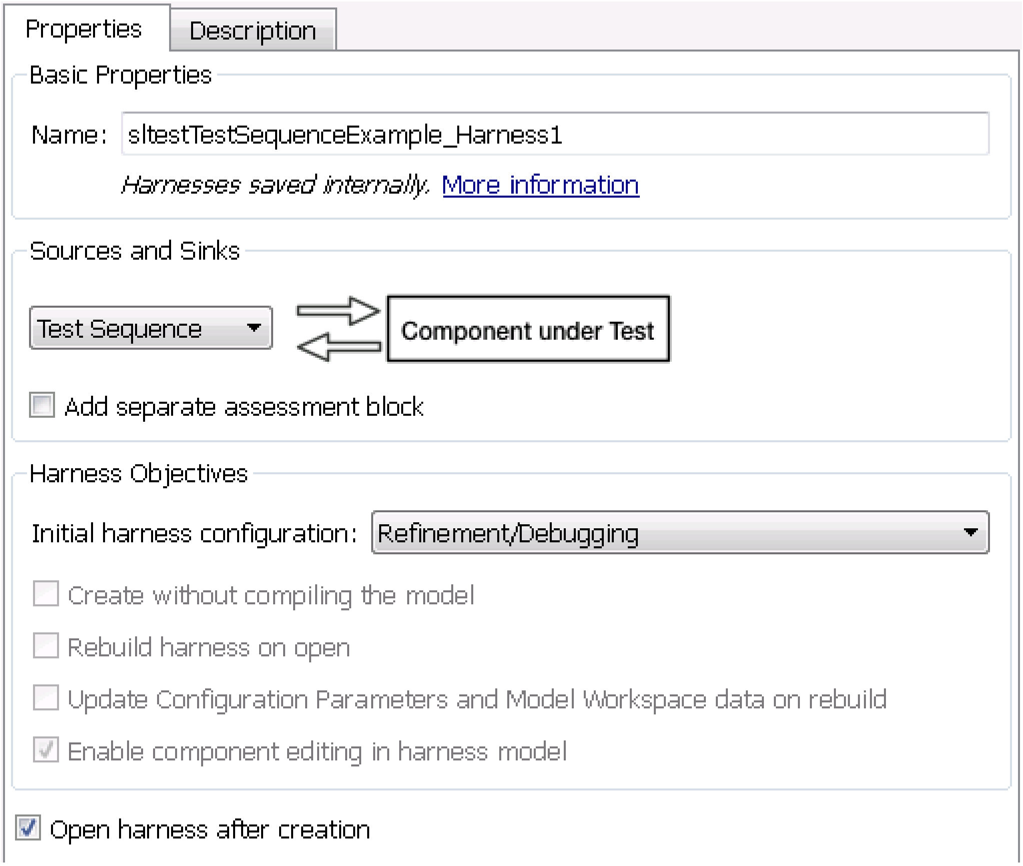
In this example, you create a simple test sequence for a transmission shift logic controller.

1. Open the model. At the command line, enter

   ```
   sltestTestSequenceExample
   ```

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**, change `Inport` to `Test Sequence`.

   The schematic displays the closed-loop configuration between the Test Sequence block and the component under test.



Name: sltestTestSequenceExample_Harness1

Harnesses saved internally. More information

Test Sequence

□ Add separate assessment block

Initial harness configuration: Refinement/Debugging

□ Create without compiling the model

□ Rebuild harness on open

□ Update Configuration Parameters and Model Workspace data on rebuild

☑ Enable component editing in harness model

☑ Open harness after creation

4 Click **OK**. The test harness for the `shift_controller` subsystem opens. Double-click the Test Sequence block.

The Test Sequence Editor opens and displays action and transition tips. Click the X to close the tips. The first line in a **Step** cell defines the step name.

5 Create the test sequence.

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

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

b Rename the second step `Stop` and add the step actions:

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

c Right-click `Accelerate` and select **Add sub-step**. Create a total of four substeps for `Accelerate`.

These steps check the component under test during the test sequence.

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

e Hover over `Limit` and click **Edit**. In the **Initial value** field, enter `2`. Click **OK**.

f In the **Transition** column, enter the transition condition for `Accelerate`. This condition uses the duration operator and transitions to the next step when the system is in fourth gear for 2 seconds.

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

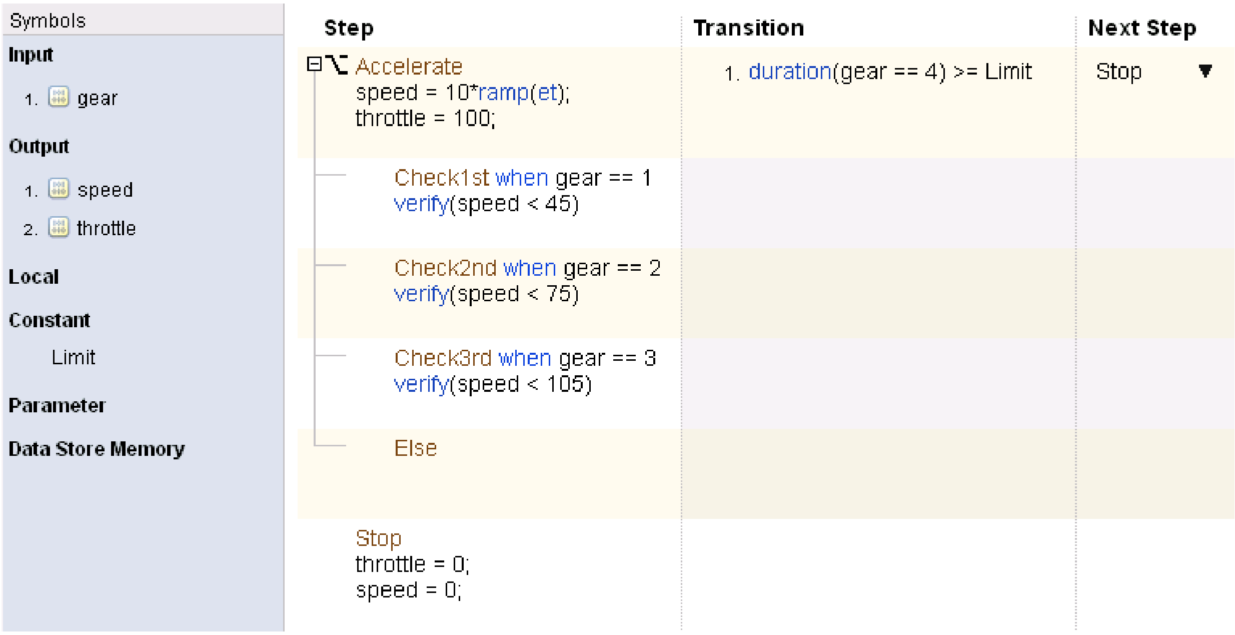
In the **Next Step** column, select `Stop`.

g Change the `Accelerate` group to a When decomposition sequence. Right-click `Accelerate` and select **When decomposition**.

h Enter the names and actions for the substeps.

```
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` takes no action. `Else` handles conditions that make no other `when` statement valid.



Symbols

Input

1. gear

Output

1. speed
2. throttle

Local

Constant

Limit

Parameter

Data Store Memory

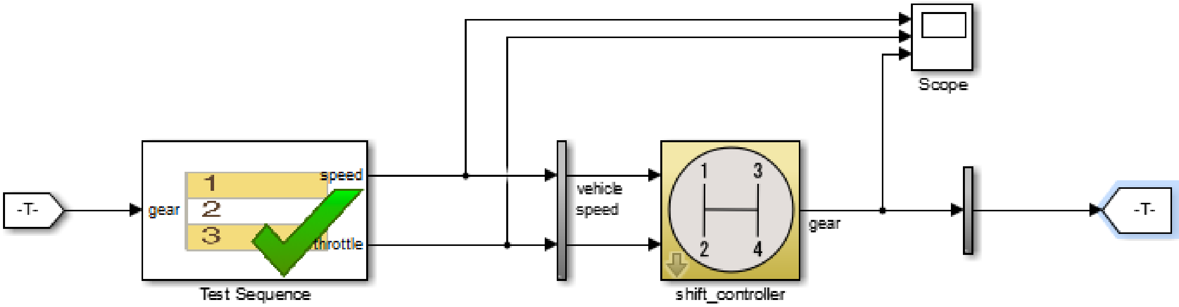
Step	Transition	Next Step

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

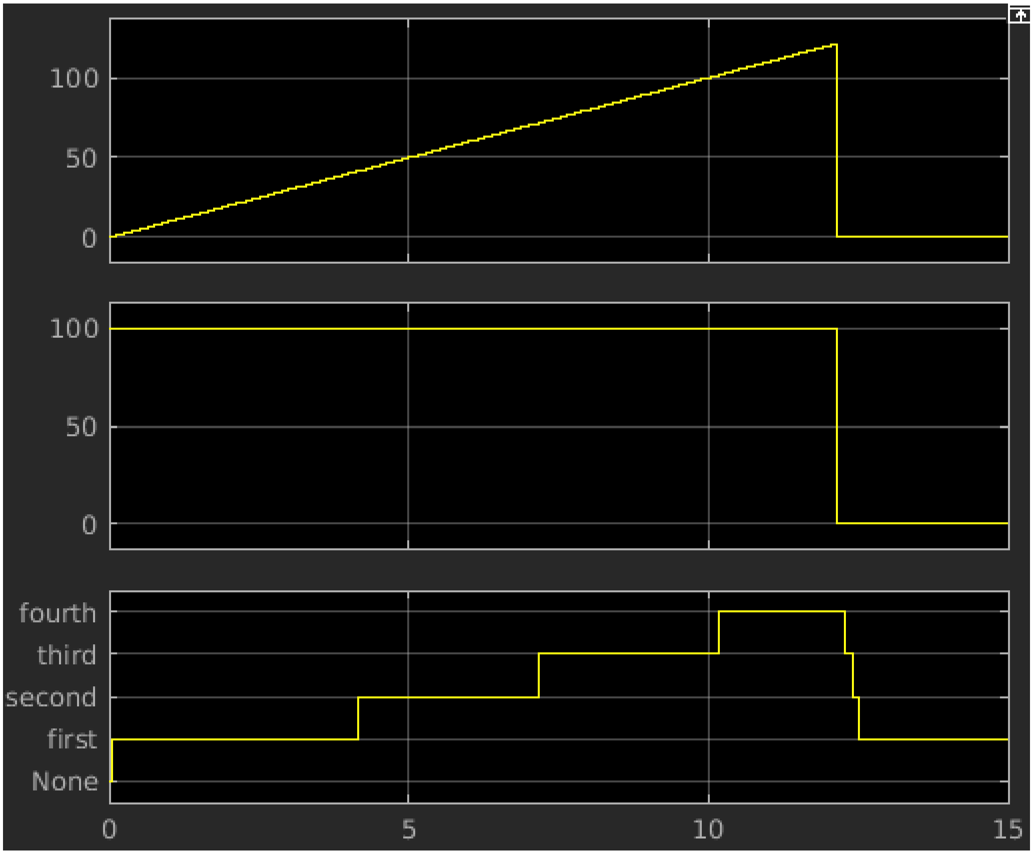
Check1st when gear == 1 verify(speed < 45)		
Check2nd when gear == 2 verify(speed < 75)		
Check3rd when gear == 3 verify(speed < 105)		
Else		

Stop
throttle = 0;
speed = 0;		

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



7 Set the model simulation time to 15 seconds and simulate the test harness.



See Also

Test Sequence

Related Examples

- "Syntax for Test Sequences and Assessments" on page 3-37
- "Programmatically Create a Test Sequence" on page 3-31

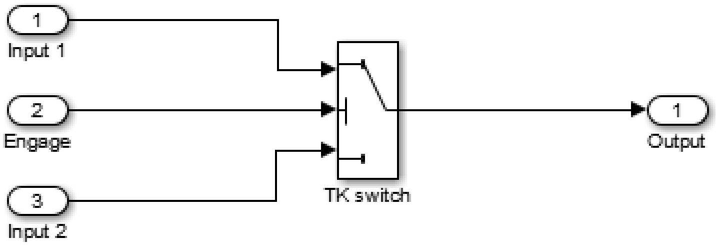
Organize Test Sequences

Compared to using timeseries data, using the Test Sequence block to define your test inputs has these advantages:

- You can organize test scenarios in test step groups, and use hierarchy levels to isolate test scenario execution.
- You can isolate model functionality by separating signal commands into distinct test steps.
- Steps can execute in response to the model, using logical conditions.
- You can author assessments for specific test conditions.
- You can concisely express signal patterns, such as waveforms, using output commands.

Before creating test steps, consider the test sequence organization. Clear organization helps communicate the test sequence intent and structure.

Consider the case of verifying a simple subsystem. The subsystem consists of a switch controlled by the `Engage` signal.



The goal of the test is to complete a simple verification of the switch function. The test does not cover all objectives for full verification, but covers a simple design check. Check that the output equals `Input 1` when the control is engaged, and `Input 2` when the control is not engaged. You organize a test sequence into an initialization step and two test scenarios. Each scenario sets `Input 1` and `Input 2`, then sets `Engage`, then assesses the switch output:

1 **Initialize the signals**

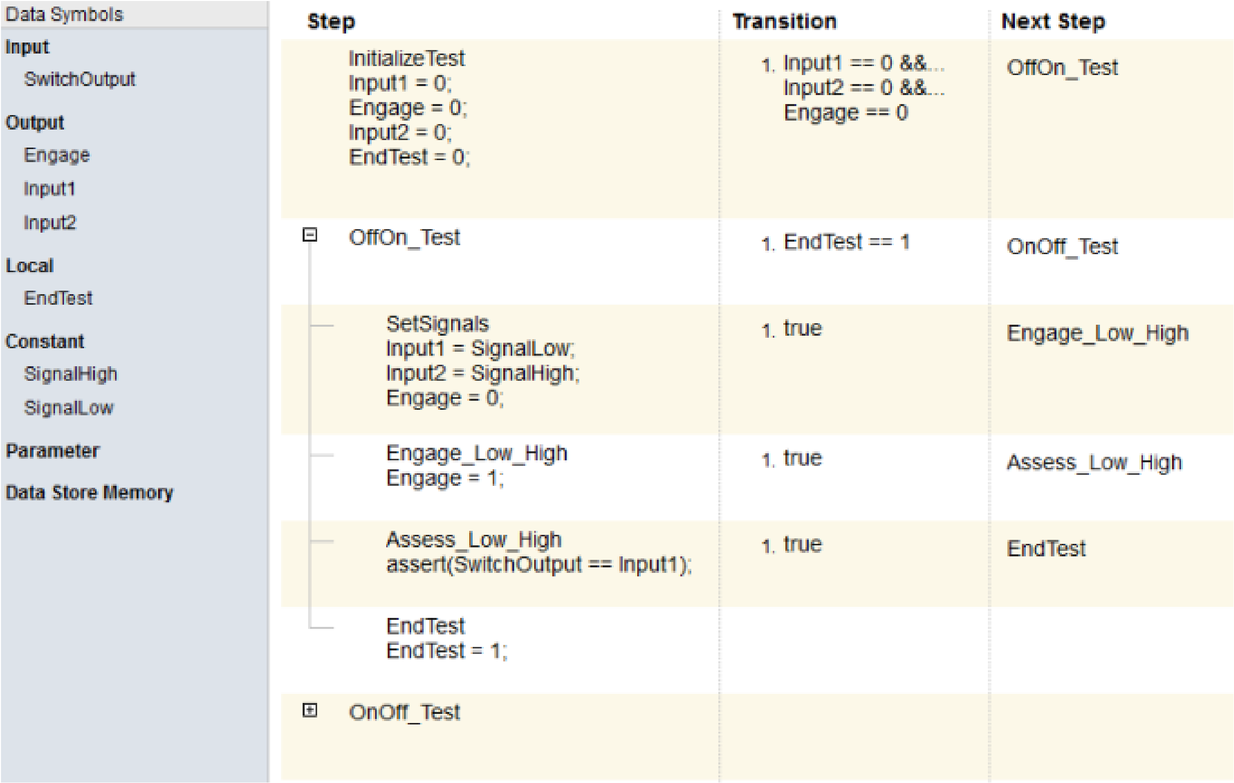
2 **Scenario 1**

- **a** Set the signal levels
- **b** Engage the control
- **c** Assess the result

3 **Scenario 2**

- **a** Set the signal levels
- **b** Engage the control
- **c** Assess the result

In the test sequence editor, the step hierarchy follows the hierarchy of the scenario outline:



Data Symbols

Input
SwitchOutput

Output
Engage
Input1
Input2

Local
EndTest

Constant
SignalHigh
SignalLow

Parameter

Data Store Memory

Step	Transition	Next Step

InitializeTest
Input1 = 0;
Engage = 0;
Input2 = 0;
EndTest = 0;
	1. Input1 == 0 &&...
	Input2 == 0 &&...
	Engage == 0
		OffOn_Test

⊟ OffOn_Test	1. EndTest == 1	OnOff_Test

SetSignals
Input1 = SignalLow;
Input2 = SignalHigh;
Engage = 0;
	1. true	Engage_Low_High

Engage_Low_High Engage = 1;	1. true	Assess_Low_High
Assess_Low_High assert(SwitchOutput == Input1);	1. true	EndTest
EndTest EndTest = 1;		
⊞ OnOff_Test		

Note: To execute test steps sequentially without using a logical transition condition, use the condition `true`. `true` moves the sequence to the next step after the current step.

Test Sequence Action and Transition Operations

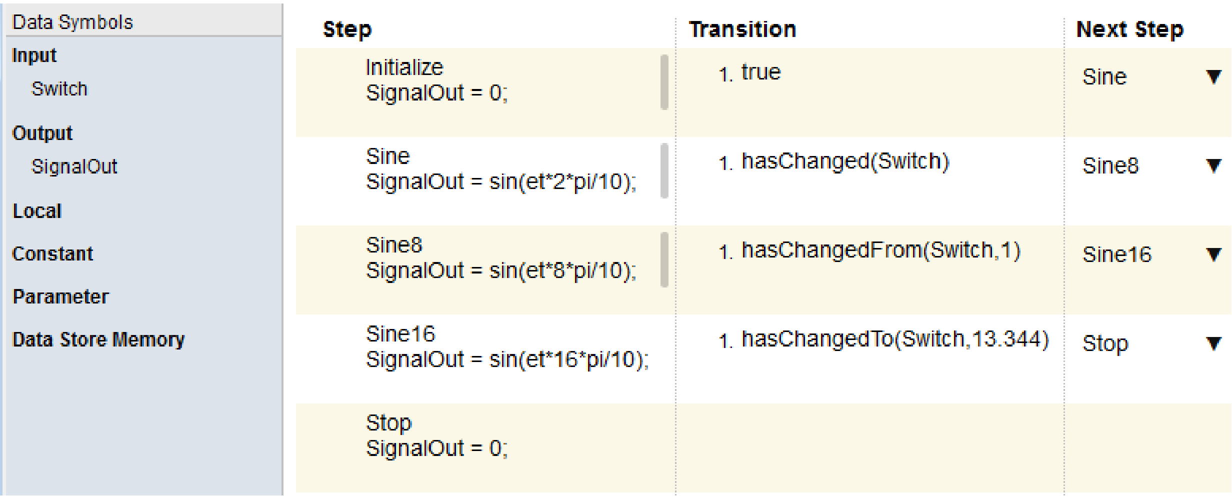
In this section...

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`



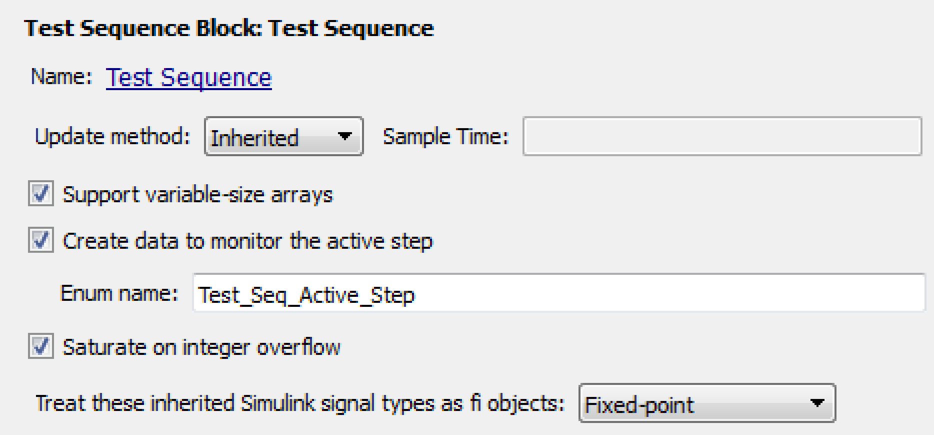
Step	Transition	Next Step
Initialize `SignalOut = 0;`	1. true	Sine ▼
Sine `SignalOut = sin(et*2*pi/10);`	1. hasChanged(Switch)	Sine8 ▼
Sine8 `SignalOut = sin(et*8*pi/10);`	1. hasChangedFrom(Switch,1)	Sine16 ▼
Sine16 `SignalOut = sin(et*16*pi/10);`	1. hasChangedTo(Switch,13.344)	Stop ▼
Stop `SignalOut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

Link a Test Assessment to an Active Test Sequence Step

If you use a separate Test Assessment block, you can link test assessments to the active test step in a Test Sequence block. You link the two blocks with data monitoring the active step:

1. Open the Model Explorer by selecting **View** > **Model Explorer** > **Model Explorer**.
2. Select the Test Sequence block in the **Model Hierarchy**.
3. In the properties, select **Create data to monitor the active step**.

   This creates a new enumerated data output. Enter a name for the enumeration.

   Test Sequence Block: Test Sequence

   Name: Test Sequence

   Update method: Inherited ▾ Sample Time:

   ☑ Support variable-size arrays

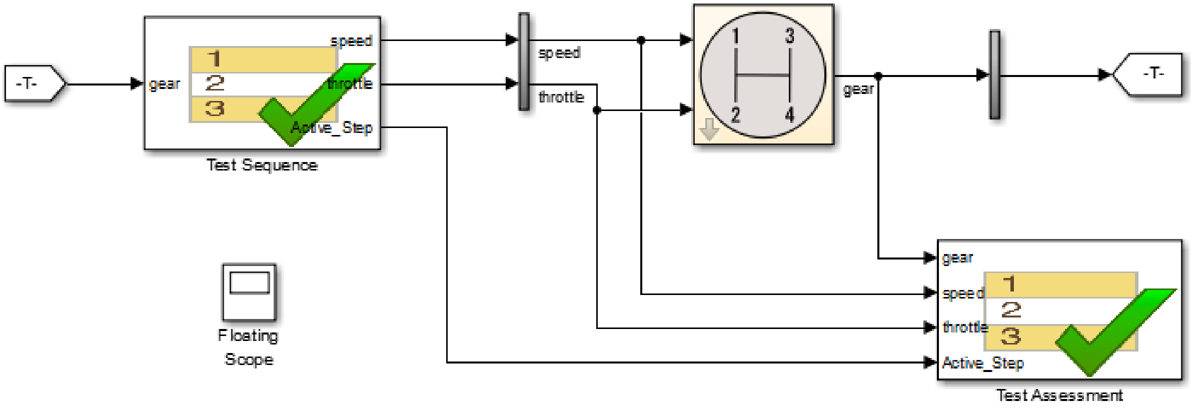
   ☑ Create data to monitor the active step

   Enum name: Test_Seq_Active_Step

   ☑ Saturate on integer overflow

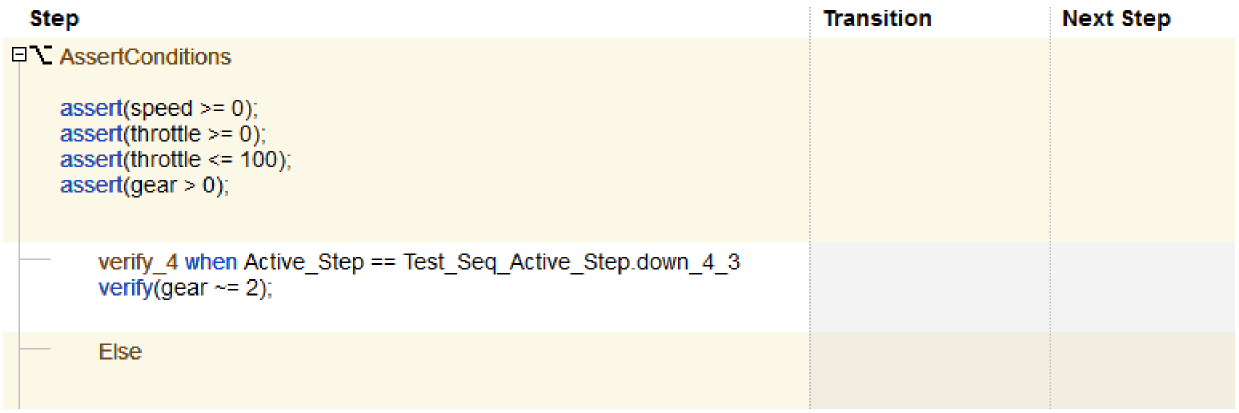
   Treat these inherited Simulink signal types as fi objects: Fixed-point ▾

4. 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. Name the input.
5. In the test harness, connect the Test Sequence block step output to the Test Assessment block step input.



6 In the test assessments, use the enumeration in actions and transitions.

For example, this Test Assessment block verifies that when the test step `down_4_3` is active, `gear ~= 2`.



Step	Transition	Next Step

⊟ AssertConditions
 	
assert(speed >= 0);
assert(throttle >= 0);
assert(throttle <= 100);
assert(gear > 0);		

verify_4 when Active_Step == Test_Seq_Active_Step.down_4_3 verify(gear ~= 2);		
Else		

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` after `Condition` becomes `true`, within the statement test step.	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)`

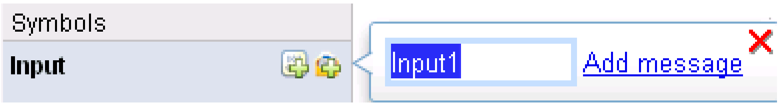
Use Messages in Test Sequences

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 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.

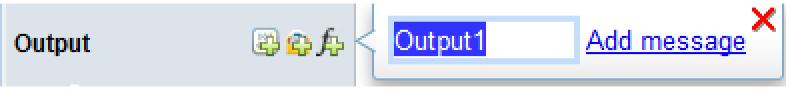


`receive(M)` determines whether a message is present in the input queue `M`, and removes the message from the queue. `receive(M)` returns `true` if a message is in the queue, and `false` if not. Once the message is received, you can access the message data using the dot notation, `M.data`, or forward 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**. For more information see Queuing Behavior of Stateflow Messages.

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

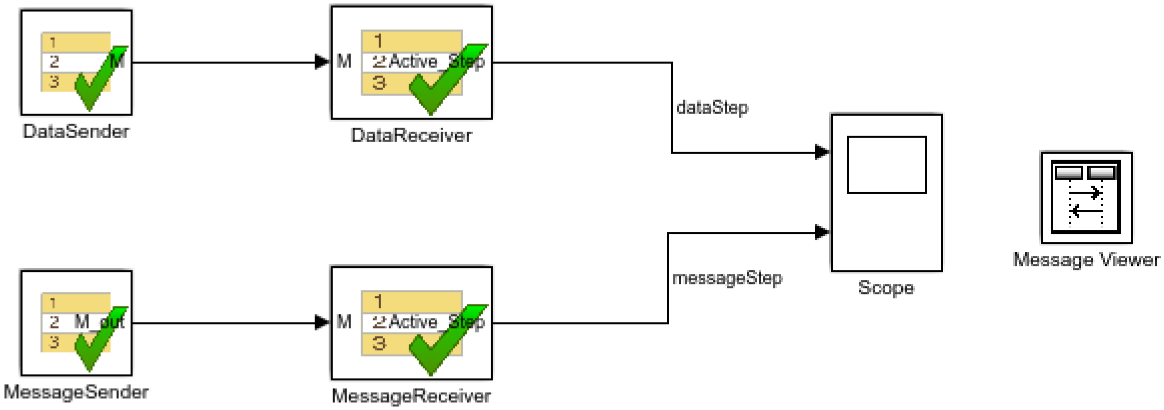
This example demonstrates message inputs and outputs, sending, and receiving a message. The model compares two pairs of test sequences. Each pair is comprised of a sending and receiving test sequence block. The first pair sends and receives data, and the second sends and receives a message.

Set the following path and model name variables.

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

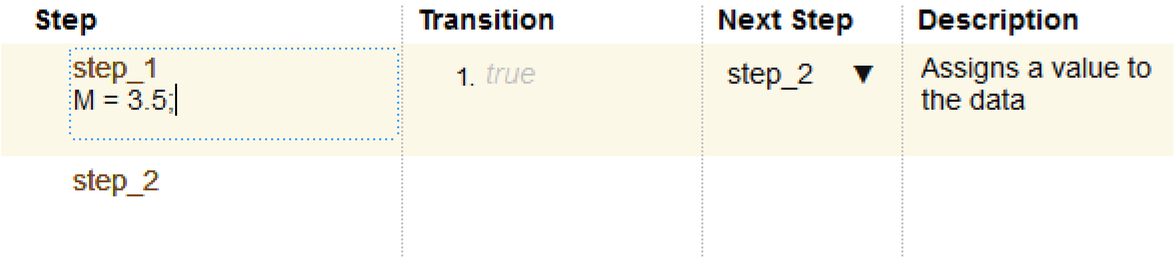
Open the model.

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



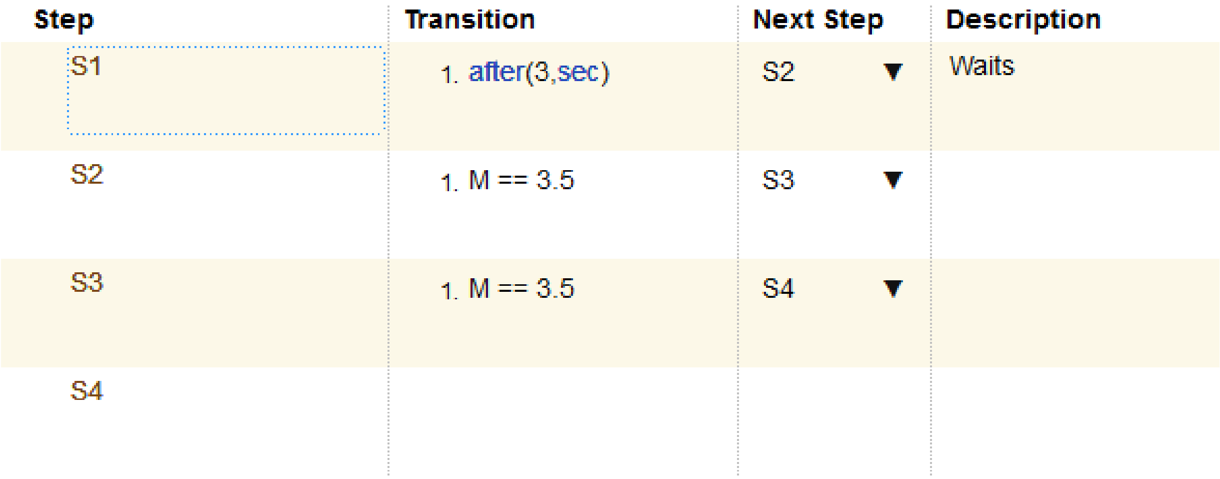
Test Sequences Using Data

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			

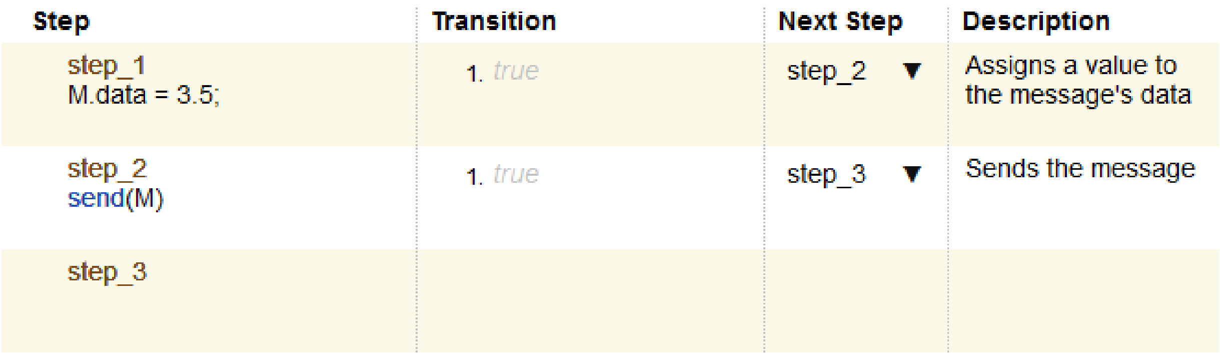
The DataReceiver block waits 3 seconds, then transitions to step S2. Step S2 transitions to step S3 using a condition comparing M to the expected value, and does the same for S3 to S4.



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			

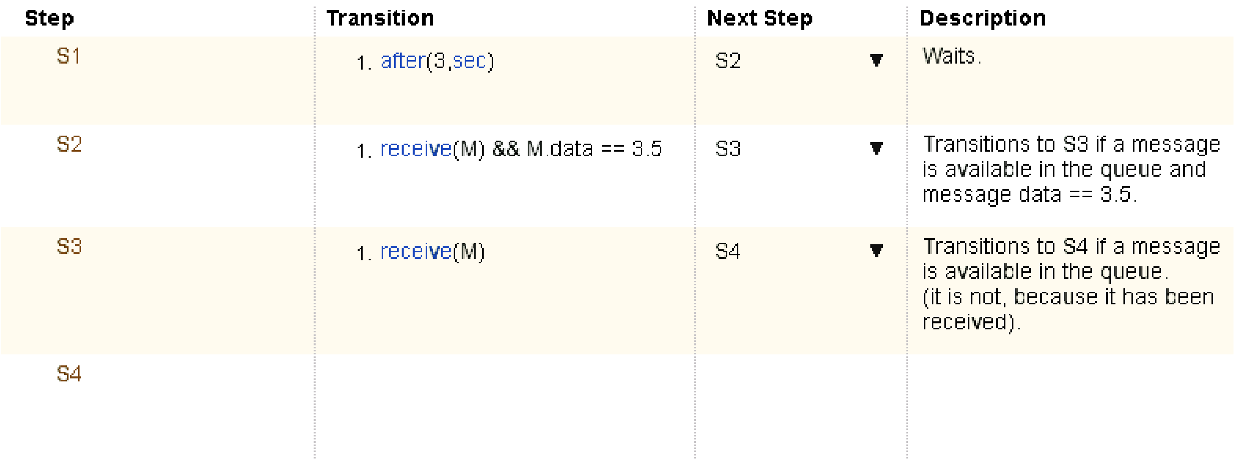
Test Sequences Using Messages

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.



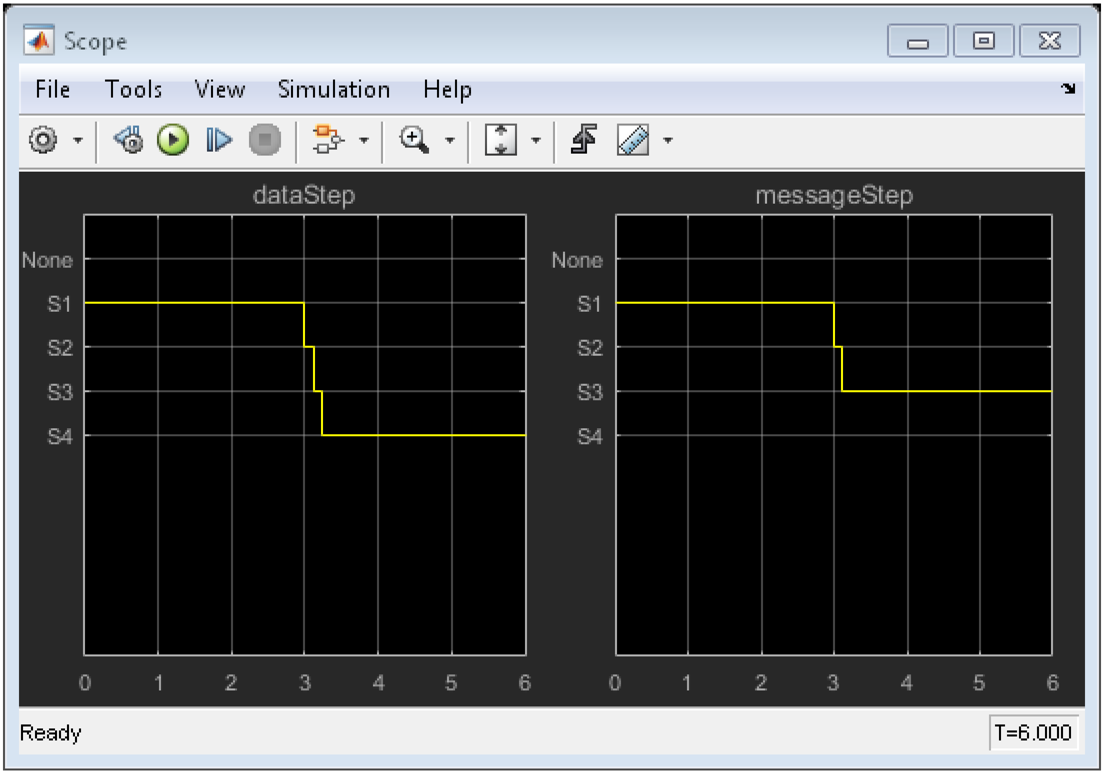
Step	Transition	Next Step	Description
S1	1. after(3,sec)	S2 ▼	Waits.
S2	1. receive(M) && M.data == 3.5	S3 ▼	Transitions to S3 if a message is available in the queue and message data == 3.5.
S3	1. receive(M)	S4 ▼	Transitions to S4 if a message is available in the queue. (it is not, because it has been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



See Also

“Syntax for Test Sequences and Assessments” on page 3-37 | Test Sequence

Related Examples

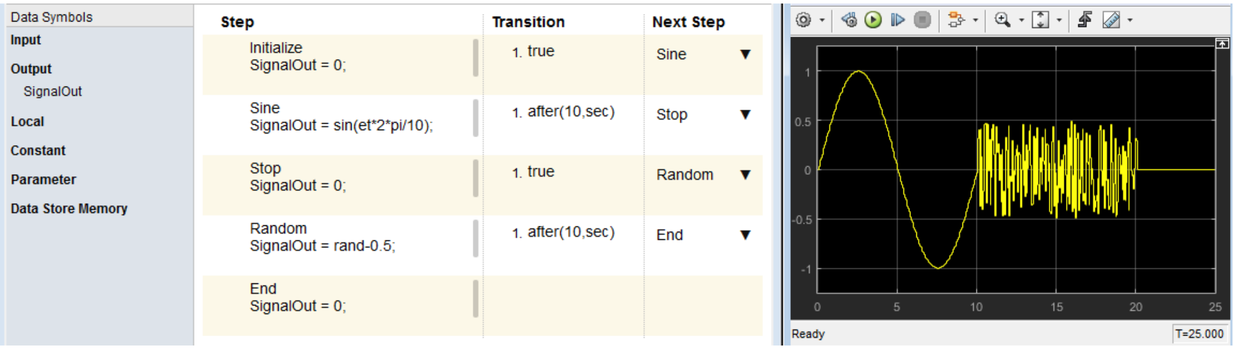
- “Generate Function-Based Test Signals” on page 3-21
- “Assess Simulation Using Logical Statements” on page 3-25

Generate Function-Based Test Signals

The Test Sequence block uses MATLAB as the action language. You can use functions to generate signal outputs to the component under test.

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.

In this test sequence, 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`.

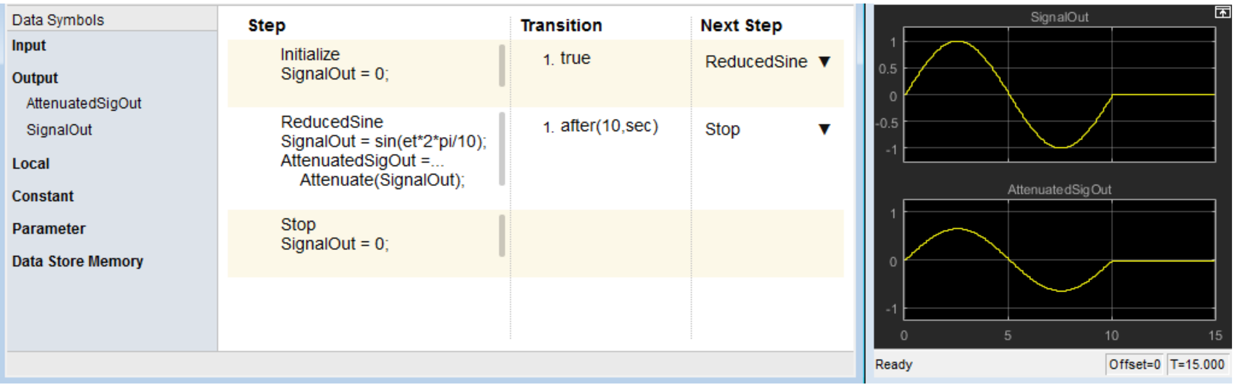


Data Symbols
Input
Output
SignalOut
Local
Constant
Parameter
Data Store Memory

Step	Transition	Next Step
Initialize SignalOut = 0;	1. true	Sine ▼
Sine SignalOut = sin(et*2*pi/10);	1. after(10,sec)	Stop ▼
Stop SignalOut = 0;	1. true	Random ▼
Random SignalOut = rand-0.5;	1. after(10,sec)	End ▼
End SignalOut = 0;		

You can also define a function in a script on the MATLAB path, and call the function in the Test Sequence block. In this test sequence, the `ReducedSine` step reduces `SignalOut` using the function `Attenuate`.

```
function[y] = Attenuate(x)
y = 0.65*x;
end
```



Data Symbols
Input
Output
AttenuatedSigOut
SignalOut
Local
Constant
Parameter
Data Store Memory

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

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

Stop SignalOut = 0;		

Output Functions

Generate test signals using output functions. The temporal operator `et` returns the elapsed time of the test step in seconds.

Note: Function outputs are not constrained to provide a defined pattern. Scaling, rounding, and other approximations of argument values can affect function outputs.

Common output 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	Output a sawtooth wave with a period of 10 sec: `sawtooth(et/10)`

Function	Syntax	Description	Example
		period of 1 and range –1 to 1. Within the interval 0 <= x < 1, sawtooth(x) increases.	
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)
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)

Function	Syntax	Description	Example
`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 number.	Generate values from the uniform distribution on the interval `[a, b]`. `a + (b—a)*rand`
`randn`	`randn`	Normally distributed pseudorandom number.	Generate values from a normal distribution with mean `1` and standard deviation `2`. `1 + 2*randn(100,1)`
`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

"Syntax for Test Sequences and Assessments" on page 3-37 | Test Sequence

Related Examples

- "Test Sequence Action and Transition Operations" on page 3-11
- "Assess Simulation Using Logical Statements" on page 3-25

Assess Simulation Using Logical Statements

In this section...

A verify statement sends results to the Test Manager and allows simulation to run even when the logical condition fails. An assert statement stops simulation. You can use verify and assert statements to assess your model.

verify

The verify keyword assesses a logical expression inside a Test Sequence or Test Assessment block. Optional arguments label results in the Test Manager and diagnostic viewer. The keyword and arguments constitute a verify statement. Use the logical expression to define a verification constraint on the system under test.

For each simulation step, the verify statement reports whether the logical expression fails, passes, or is untested. For an overall test, a verify statement returns an overall fail, pass, or untested result. Any failure at a simulation step results in an overall failure. If the verify statement does not fail, and at any time the statement passes, the overall result passes. Otherwise, the statement is not tested, and the overall result is untested. Review results in the **Verify Statements** section of the Test Manager.

Syntax

A verify statement uses syntax of these forms

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

The simplest verify statement uses only a logical expression. To make results easier to interpret, use additional arguments to define an error message 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'
```

Continuous-Time Considerations

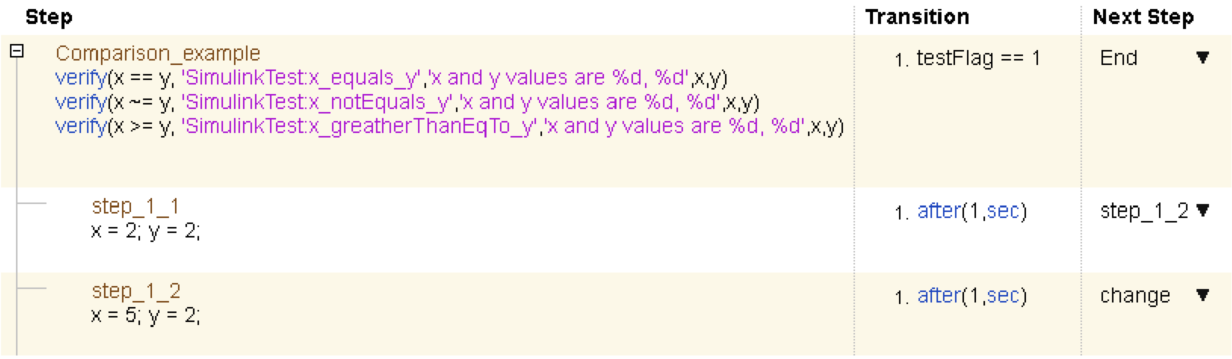
`verify` is not supported in Test Sequence blocks that use continuous-time updating. Test Sequence block data can depend on factors such as the solver step time. Continuous-time updating can cause differences in when block data and `verify` statements update, which can lead to unexpected `verify` statement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

Example

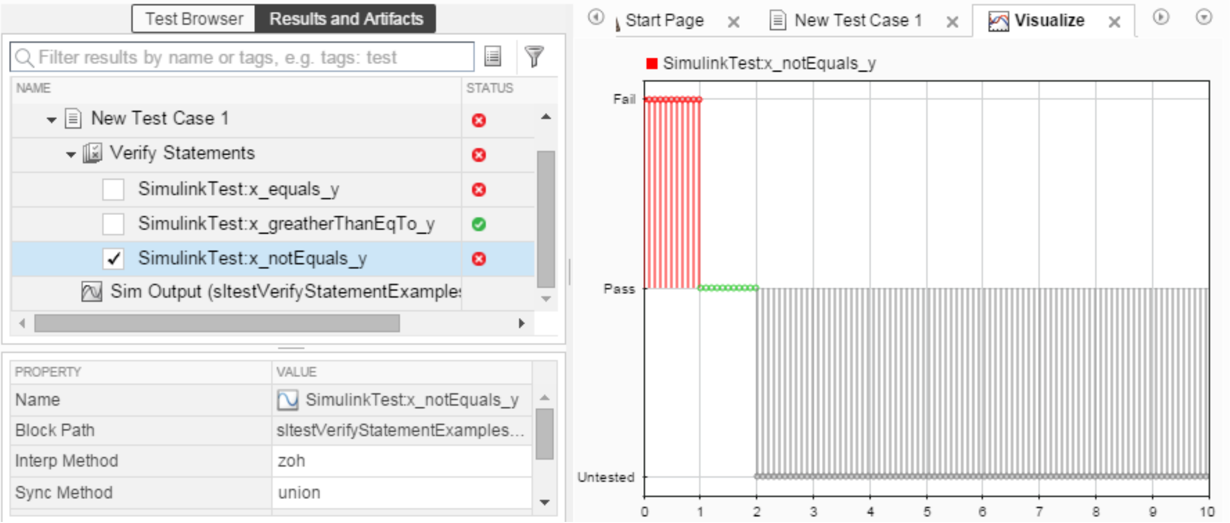
In this comparison of two values, the parent step uses `verify` statements to assess two local variables `x` and `y` during the simulation. The substeps set two conditions.

- `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.



Step	Transition	Next Step
⊟ 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)	1. testFlag == 1	End ▼
step_1_1 x = 2; y = 2;	1. after(1,sec)	step_1_2 ▼
step_1_2 x = 5; y = 2;	1. after(1,sec)	change ▼

The Test Manager displays the results.



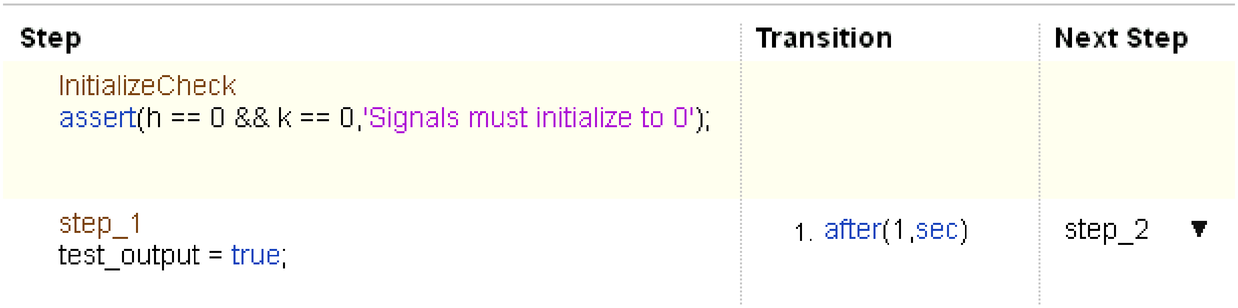
NAME	STATUS
New Test Case 1	☒
Verify Statements	☒
□ SimulinkTest:x_equals_y	☒
□ SimulinkTest:x_greatherThanEqTo_y	✓
☑ SimulinkTest:x_notEquals_y	☒
Sim Output (sltestVerifyStatementExample:	

PROPERTY	VALUE
Name	SimulinkTest:x_notEquals_y
Block Path	sltestVerifyStatementExamples...
Interp Method	zoh
Sync Method	union

assert

`assert` evaluates a logical argument, but unlike `verify`, `assert` stops simulation. `assert` does not return fail, pass, or untested results. Failures appear as errors. Consider using `assert` statements to avoid executing a bad test. For example, if a component under test outputs two signals `h` and `k`, and the test requires `h` and `k` initialized to `0`, use `assert` to stop the test if the signals do not initialize.

To make results easier to interpret, add an optional message that evaluates when the assertion fails. This example demonstrates an `assert` statement that returns a message if the logical condition 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

Code is not generated for `assert` statements in the Test Sequence block.

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`

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

"Syntax for Test Sequences and Assessments" on page 3-37 | Test Sequence

Related Examples

- "Test Sequence Action and Transition Operations" on page 3-11
- "Generate Function-Based Test Signals" on page 3-21

Programmatically Create a Test Sequence

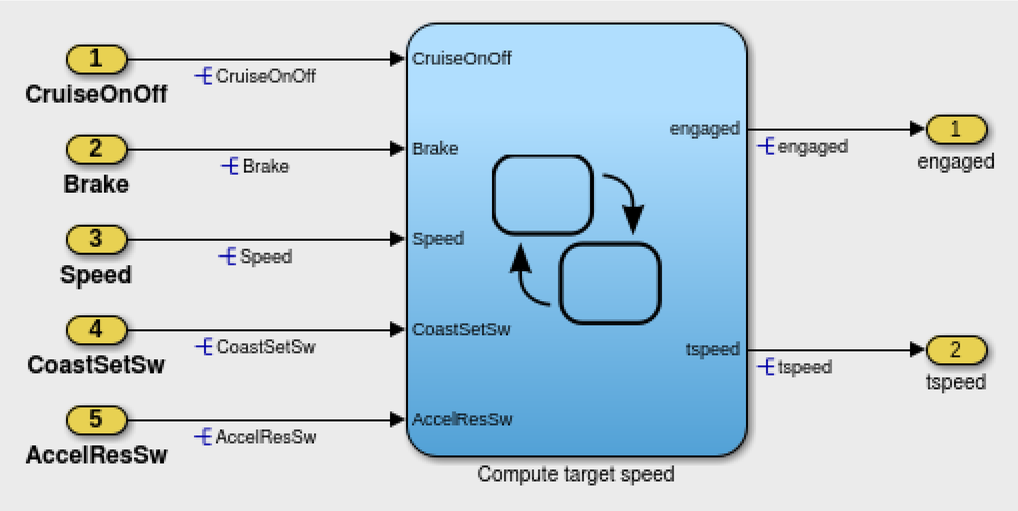
This example shows how to create a test sequence programmatically. You create a Test Sequence block, and author a test sequence to verify two safety requirements of a cruise control system.

Create a Test Harness Containing a Test Sequence Block

1. Open the cruise control project, and open the model. This creates a working copy of the project in your MATLAB folder.

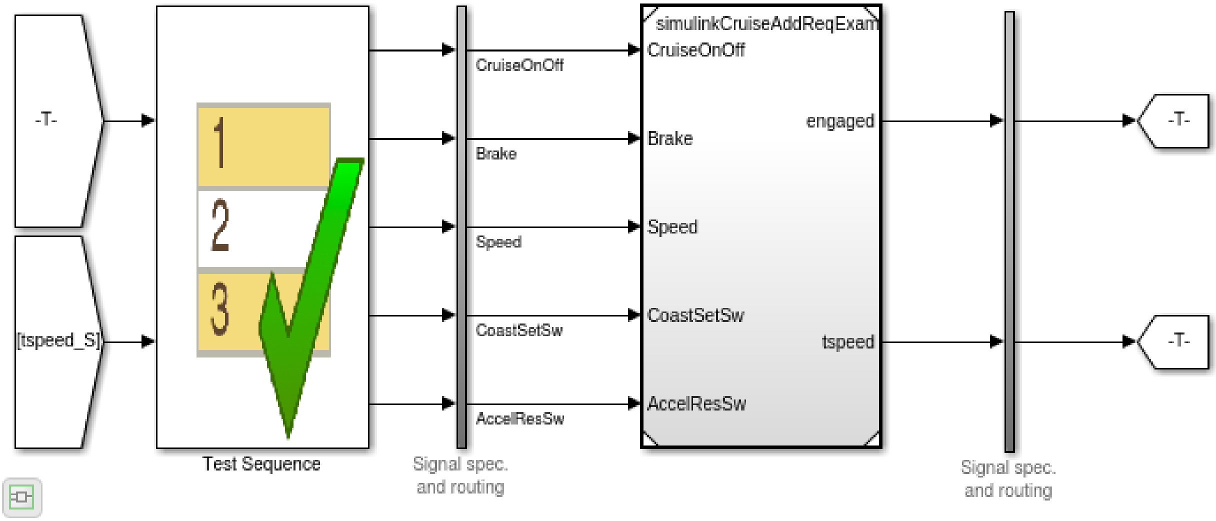
```
slVerificationCruiseStart;
open_system simulinkCruiseAddReqExample.slx
```

```
Initializing: Project Path
Identifying shadowed project files
```



2. Create and open the test harness.

```
sltest.harness.create('simulinkCruiseAddReqExample','Name','SafetyTestHarness',...
    'Source','Test Sequence')
sltest.harness.open('simulinkCruiseAddReqExample','SafetyTestHarness')
set_param('SafetyTestHarness','StopTime','15');
```



Author the Test Sequence

1. Add a local variable `endTest`, which you use to transition between test steps.

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

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

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

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

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

    % Add a transition from |Initialize1| to |BrakeTest|.
    sltest.testsequence.addTransition('SafetyTestHarness/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 = single(50);');
    sltest.testsequence.addStep('SafetyTestHarness/Test Sequence',...
        'BrakeTest.SetValues','Action',setValuesActions)

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

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

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

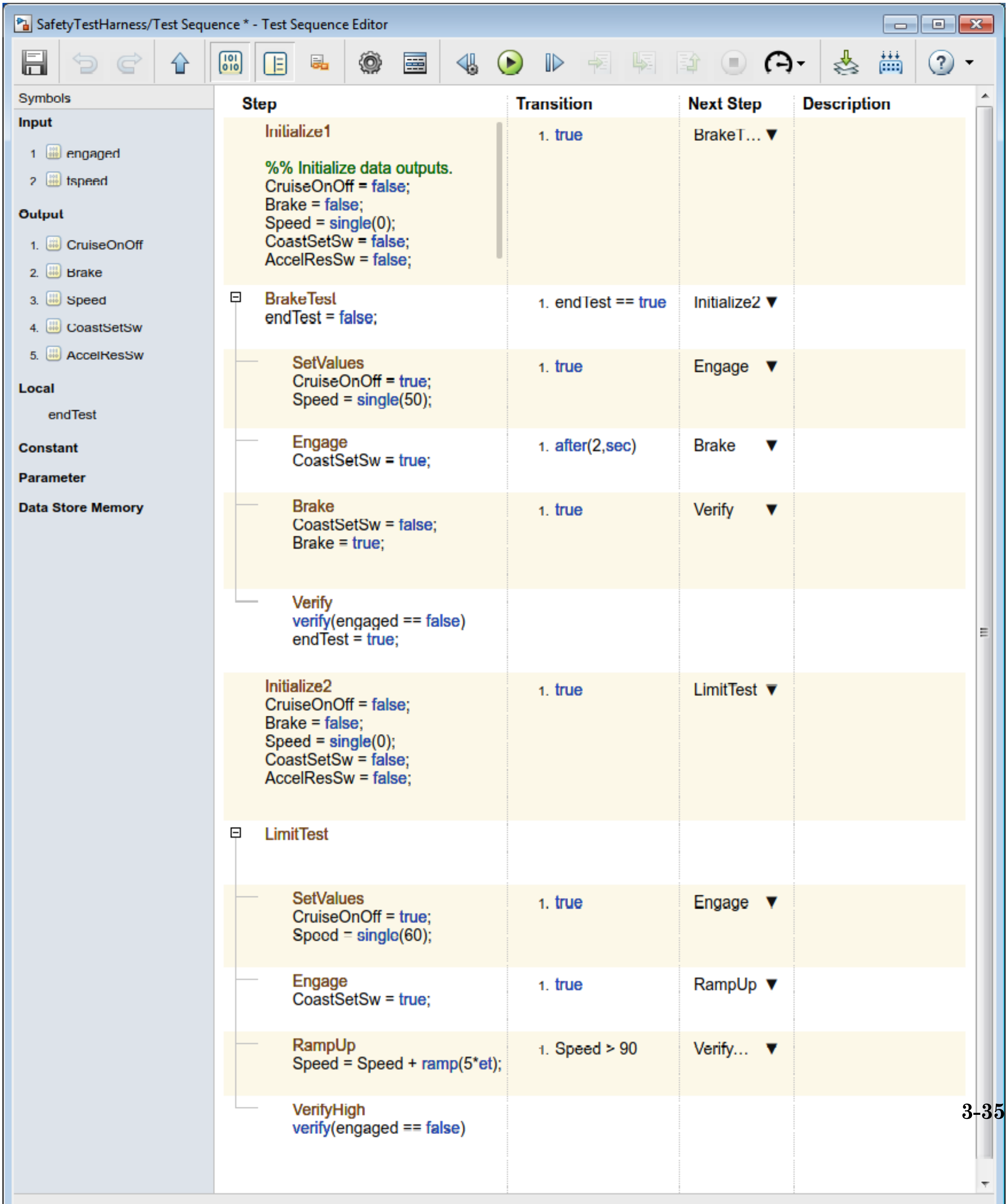
4. Add a step `Initialize2` to initialize component inputs again, and add a transition from `BrakeTest` to `Initialize2`.

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

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

```
sltest.testsequence.addStepAfter('SafetyTestHarness/Test Sequence',...
    'LimitTest','Initialize2')
sltest.testsequence.addTransition('SafetyTestHarness/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('SafetyTestHarness/Test Sequence',...
        'LimitTest.SetValues','Action',setValuesActions2)

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

    % Add a step to ramp the vehicle speed.
    sltest.testsequence.addStepAfter('SafetyTestHarness/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('SafetyTestHarness/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('SafetyTestHarness/Test Sequence',...
        'LimitTest.SetValues','true','LimitTest.Engage')
    sltest.testsequence.addTransition('SafetyTestHarness/Test Sequence',...
        'LimitTest.Engage','true','LimitTest.RampUp')
    sltest.testsequence.addTransition('SafetyTestHarness/Test Sequence',...
        'LimitTest.RampUp','Speed > 90','LimitTest.VerifyHigh')
```

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



SafetyTestHarness/Test Sequence * - Test Sequence Editor

Symbols

Input

1 engaged

2 tspeed

Output

1. CruiseOnOff
2. Brake
3. Speed
4. CoastSetSw
5. AccelResSw

Local

endTest

Constant

Parameter

Data Store Memory

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 = single(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 = single(0);
CoastSetSw = false;
AccelResSw = false;
	1. true	LimitTest ▼	

⊟ LimitTest			

SetValues
CruiseOnOff = true;
Speed = single(60);
	1. true	Engage ▼	

Engage CoastSetSw = true;	1. true	RampUp ▼	
RampUp Speed = Speed + ramp(5*et);	1. Speed > 90	Verify… ▼	
VerifyHigh verify(engaged == false)			

Close the Test Harness and Model

```
sltest.harness.close('simulinkCruiseAddReqExample','SafetyTestHarness');
close_system('simulinkCruiseAddReqExample.slx',0);
```

Syntax for Test Sequences and Assessments

In this section...
“Assessment Statements” on page 3-28
“Temporal Operators” on page 3-13
“Transition Operators” on page 3-15
“Output Functions” on page 3-22
“Logical Operators” on page 3-29
“Relational Operators” on page 3-29

This topic describes syntax used in Test Sequence and Test Assessment block actions, transitions, and assessments. You can also work with Test Sequence blocks using the test sequence API, which provides functions to create, read, edit, and delete test sequence steps, transitions, and data symbols. See the functions listed in the **Test Sequence Programming** section on the “Logic-Based Testing” page.

Within test step actions, transitions, and assessments, Test Sequence and Test Assessment blocks use MATLAB as the action language. In addition to the MATLAB language, the block includes keywords and operators to create action, transition, and assessment statements. For example:

- Output a square wave with a period of `10` sec:

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

  ```
  hasChangedTo(h,0)
  ```
- 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)`	Assesses a logical expression.	`verify(x > y,...`
			`'SimulinkTest:greaterThan',...`

Keyword	Statement Syntax	Description	Example

	`verify(expression, errorMessage)`
	`verify(expression, identifier, errorMessage)`
		Optional arguments label results in the Test Manager and diagnostic viewer.	`'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)`
`duration`	`ElapsedTime = duration (Condition, TimeUnits)`	Returns `ElapsedTime` in `TimeUnits` after `Condition` becomes `true`, within the statement test step.	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.	Transition when `h` changes from `1`: `hasChangedFrom(h,1)`

Operator	Syntax	Description	Example
		u must be an input data symbol.	
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)

Output Functions

Generate test signals using output functions. The temporal operator et returns the elapsed time of the test step in seconds.

Note: Function outputs are not constrained to provide a defined pattern. Scaling, rounding, and other approximations of argument values can affect function outputs.

Common output 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 < 0.5and –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.	Output a sawtooth wave with a period of 10 sec: sawtooth(et/10)

Function	Syntax	Description	Example
		Within the interval `0 <= x < 1`, `sawtooth(x)` increases.	
`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)`
`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)`

Function	Syntax	Description	Example
`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 number.	Generate values from the uniform distribution on the interval `[a, b]`. `a + (b–a)*rand`
`randn`	`randn`	Normally distributed pseudorandom number.	Generate values from a normal distribution with mean `1` and standard deviation `2`. `1 + 2*randn(100,1)`
`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	`verify((p && q) || (~p && ~q))`

Operation	Syntax	Description	Example
		equivalent to biconditional `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)`

Related Examples

- “Assess Simulation Using Logical Statements” on page 3-25
- “Test Sequence Action and Transition Operations” on page 3-11
- “Generate Function-Based Test Signals” on page 3-21
- “Programmatically Create a Test Sequence” on page 3-31

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.

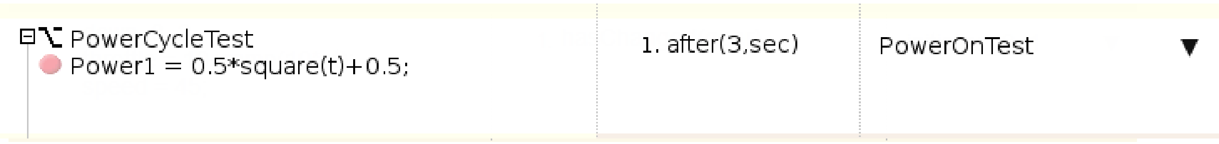
Animation speed affects simulation speed. If you slow down animation speed for debugging, return the speed to **Fast** or **Lightning Fast** when you finish debugging to avoid slowing your simulation. If you do not need the test step highlights and want the fastest simulation, choose **None**.

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.

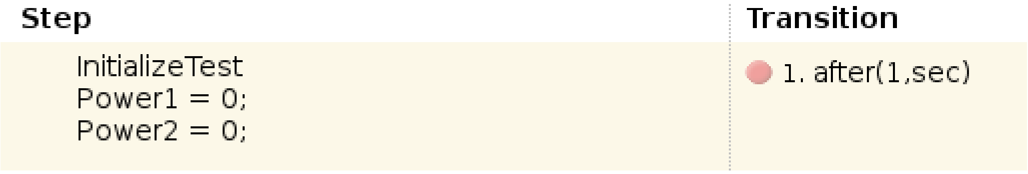
You can add breakpoints to test step actions or transitions:

- 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**.

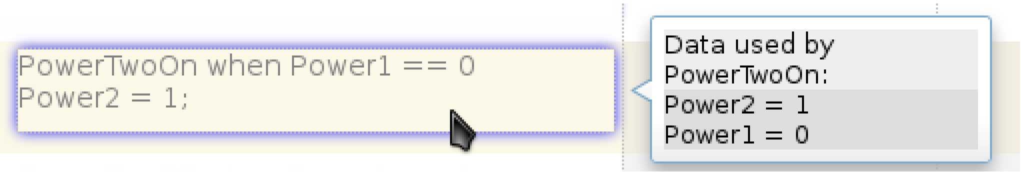


```
InitializeTest
Power1 = 0;
Power2 = 0;
```

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

Test a Model Component Using Signal Functions

In this section...

Using the Test Sequence block, you can define a set of input functions to test your component, and conditionally switch the function based on component signals. See Test Sequence for more information.

This example demonstrates building and simulating a test sequence using ramp and square wave signals. The test initializes at constant temperature, ramps down to a limit, and executes a square-wave temperature cycle.

Create a Test Sequence

1 Access the model. Enter

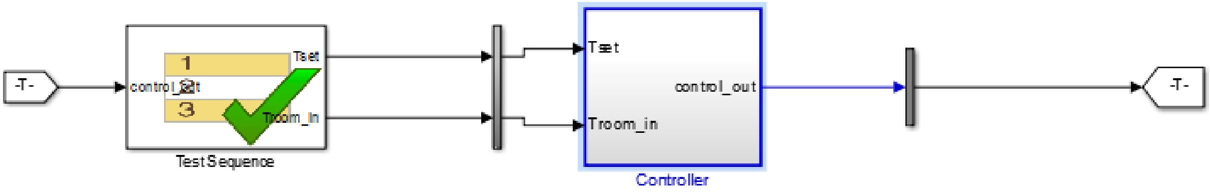
```
cd(fullfile(docroot,'toolbox','sltest','examples'))
```

2 Copy this model file and supporting files to a writable location on the MATLAB path:

```
sltestSignalFunctionExample.slx
sltestHeatpumpBusPostLoadFcn.mat
PumpDirection.m
```

3 Open the model, and open the harness.

```
open_system('sltestSignalFunctionExample');
sltest.harness.open('sltestSignalFunctionExample/Controller','RampSquareHarness')
```



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

5 Rename the first and second steps. Delete the default names and replace them with `const_90` and `ramp_down`.

6 Add a third step to the table. Right-click the `ramp_down` line, and select **Add step after**. Name the third step `temp_step`.

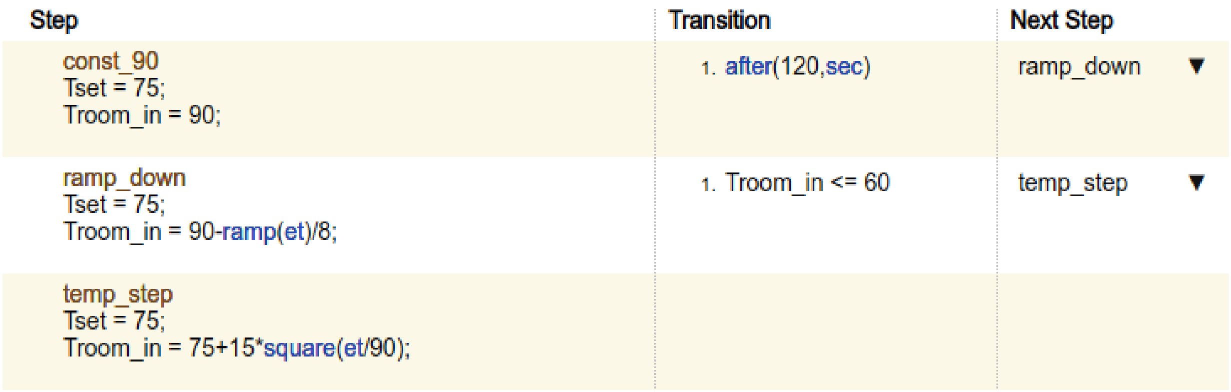
7 Add output conditions and transition fields to the steps. Copy and paste the listings from the table.

Step	Transition	Next step

`const_90`
`Tset = 75;`
`Troom_in = 90;`
	`after(120,sec)`	`ramp_down`

`ramp_down`
`Tset = 75;`
`Troom_in = 90-ramp(et)/8;`
	`Troom_in <= 60`	`temp_step`

`temp_step`
`Tset = 75;`
`Troom_in = 75+15*square(et/90);`		



Step	Transition	Next Step

const_90
Tset = 75;
Troom_in = 90;
	1. after(120,sec)	ramp_down ▼

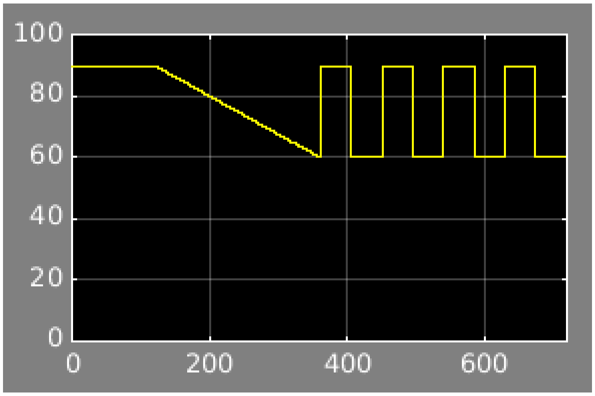
ramp_down
Tset = 75;
Troom_in = 90-ramp(et)/8;
	1. Troom_in <= 60	temp_step ▼

temp_step
Tset = 75;
Troom_in = 75+15*square(et/90);		

Simulate the Test Harness

1 Set the simulation time to 720 sec.

2 Simulate the Test Harness. Observe the `Troom_in` signal in the scope.



See Also

Blocks

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.

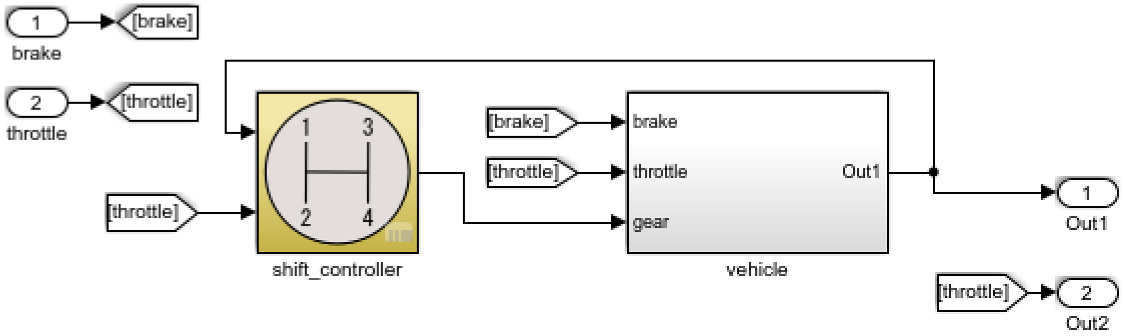
The Model and Controller

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.

```
path = fullfile(matlabroot,'examples','simulinktest');
mdl = 'TransmissionDownshiftTestSequence';
harness = 'controller_harness';
open_system(fullfile(path,mdl));
```



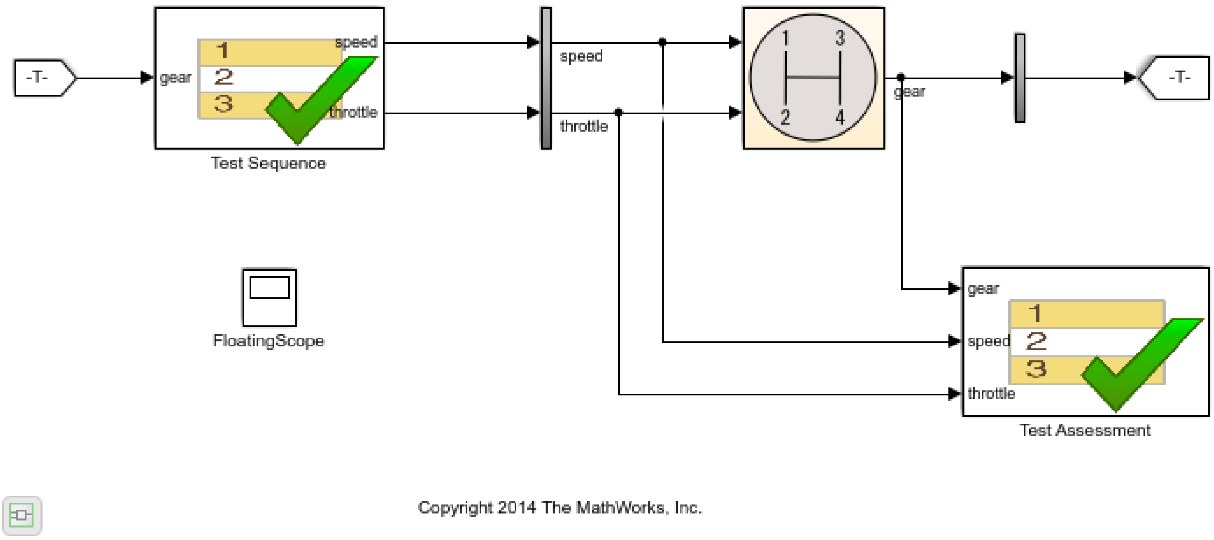
Testing Downshift Points of a Transmission Controller

Copyright 2014 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.

```
sltest.harness.open([mdl '/shift_controller'],harness)
```

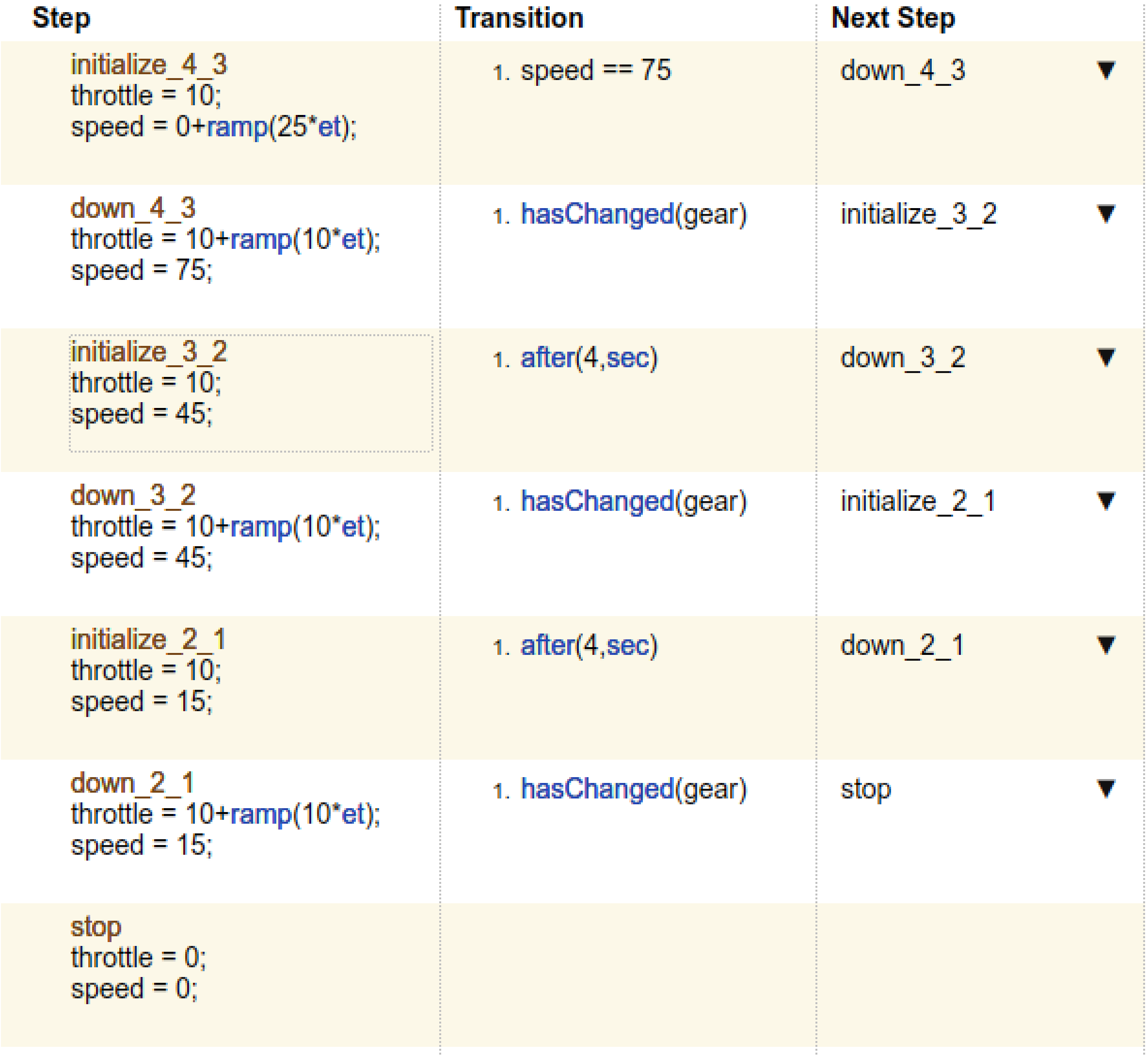


The Test Sequence

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.

```
open_system([harness '/Test Sequence']);
```



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

Assume that the requirements for the shift controller include:

- Speed shall never be negative.

- Gear shall always be positive.
- Throttle shall be between 0% and 100%.
- The controller shall not let the engine overspeed.

Open the Test Assessment block. These assertions in the block correspond to the first three requirements. 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 requirement has three sub-requirements. We assume that the engine cannot overspeed in fourth (top) gear.

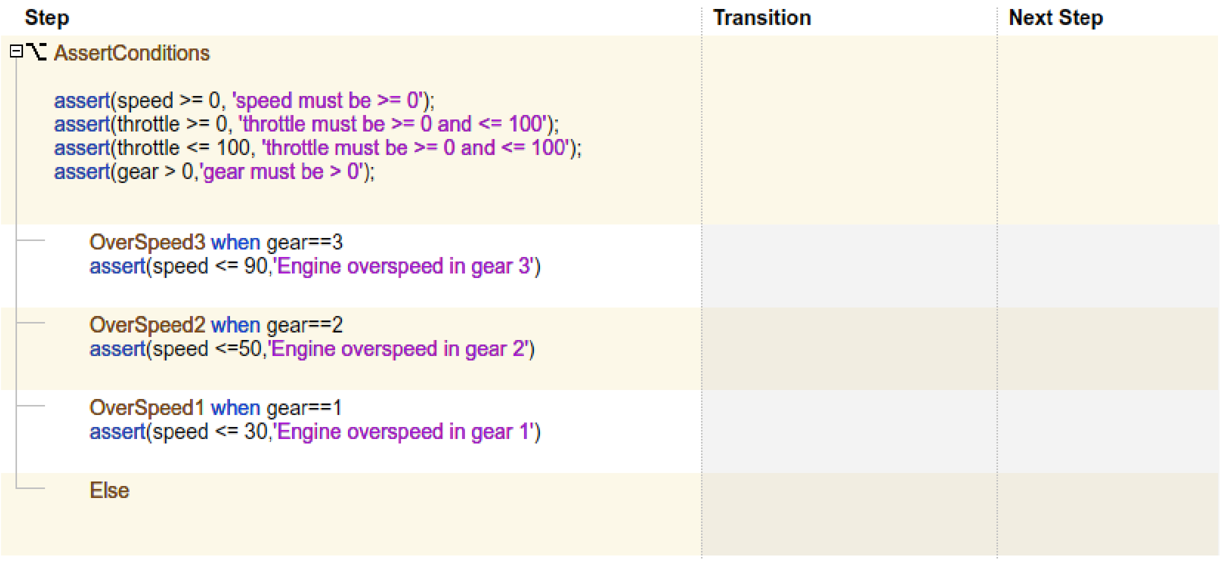
- The controller shall not let the vehicle speed exceed 90 in gear 3.
- The controller shall not let the vehicle speed exceed 50 in gear 2.
- The controller shall not let the vehicle speed exceed 30 in gear 1.

You can model these assessments with a When decomposition sequence. When decomposition step selection is based on signal conditions defined in the **Step** column, with each condition preceded by the `when` operator. The **Transition** and **Next Step** columns do not affect the transition. The last step `Else` in the when decomposition covers any undefined condition and does not use a `when` declaration.

To change a sequence to a When decomposition, right-click a step and select **When decomposition**. Sub-steps of this step then operate using the `when` operator.

`AssertConditions` has sub-steps that assess the controller as follows:

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

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

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



Step	Transition	Next Step

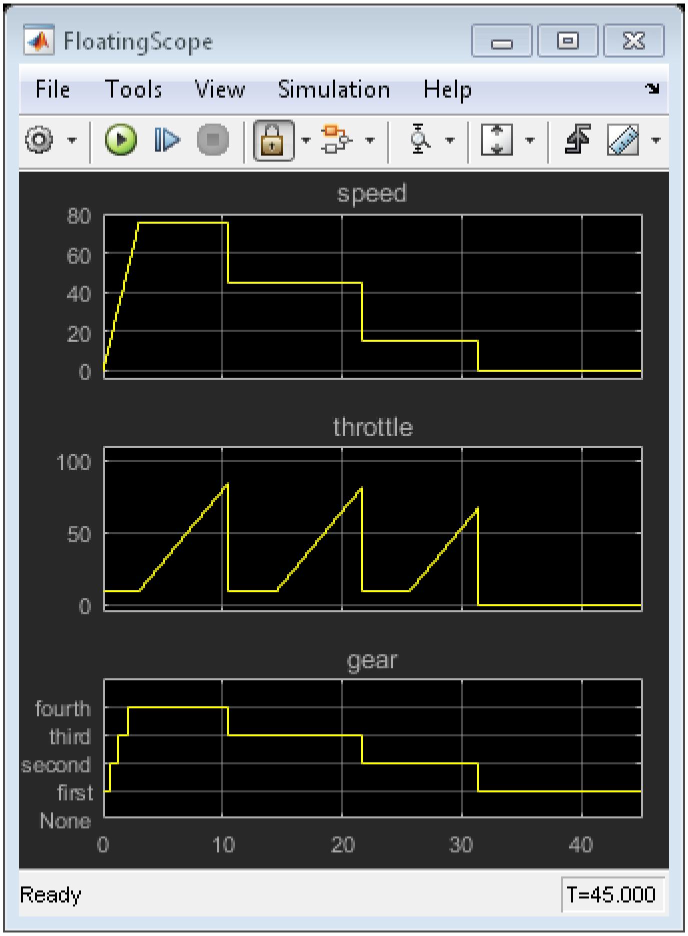
AssertConditions
 	
assert(speed >= 0, 'speed must be >= 0');
assert(throttle >= 0, 'throttle must be >= 0 and <= 100');
assert(throttle <= 100, 'throttle must be >= 0 and <= 100');
assert(gear > 0,'gear must be > 0');		

OverSpeed3 when gear==3 assert(speed <= 90,'Engine overspeed in gear 3')		
OverSpeed2 when gear==2 assert(speed <=50,'Engine overspeed in gear 2')		
OverSpeed1 when gear==1 assert(speed <= 30,'Engine overspeed in gear 1')		
Else		

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.

```
open_system([harness '/FloatingScope'])
sim(harness);
```



```
close_system(mdl);
```

Reuse Test Assessments

If one test assessment covers many test cases, consider reusing the assessment from a single source such as a library. Reusing test assessments allows you to update and manage the source rather than multiple copies of the same assessment. Often, such assessments are associated with broad requirements such as:

- "The `speed` signal must never be negative."
- "The cruise control must never be engaged while the brake is engaged."
- "The heat pump must wait more than 5 seconds before switching from on to off or off to on."
- "The projector temperature must never exceed 65 degrees Celsius."

Reuse Test Assessments Using a Library

This example shows how to reuse test assessments contained in a test sequence block using a linked block from a library.

When you create a test harness, you can include a standalone Test Sequence block for test assessments (a Test Assessment block). Often, assessments cover multiple test cases, making it convenient to reuse the same Test Assessment block. Test assessment reuse has these advantages:

- Assessments are stored in a single source. If the requirements change, you update only the assessments in the library.
- You can link to test requirements from the source. Linking from the source reduces the number of requirements links to manage.

To reuse a standalone Test Assessment block in multiple test harnesses, create the Test Assessment block in a library, and reuse the Test Assessment block in multiple test harnesses by way of linked blocks.

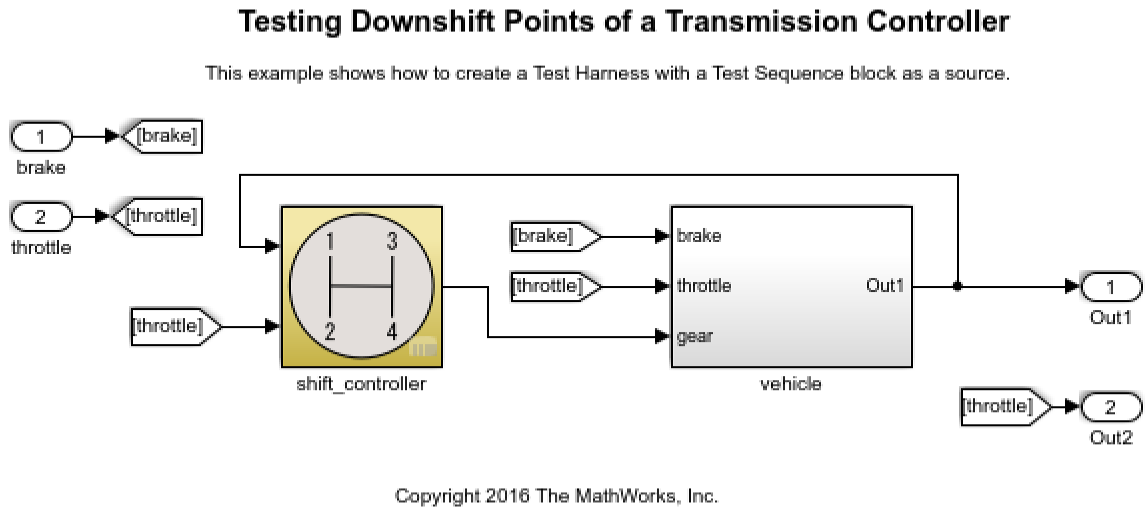
Consider using a library for high-level test assessments that correspond to multiple test cases.

You can also create reusable assessments in a library using blocks from the Model Verification library in Simulink.

Explore the Test Sequence Example Model

1. Open the model. At the command line, enter:

```
sltestTestSequenceExample
```

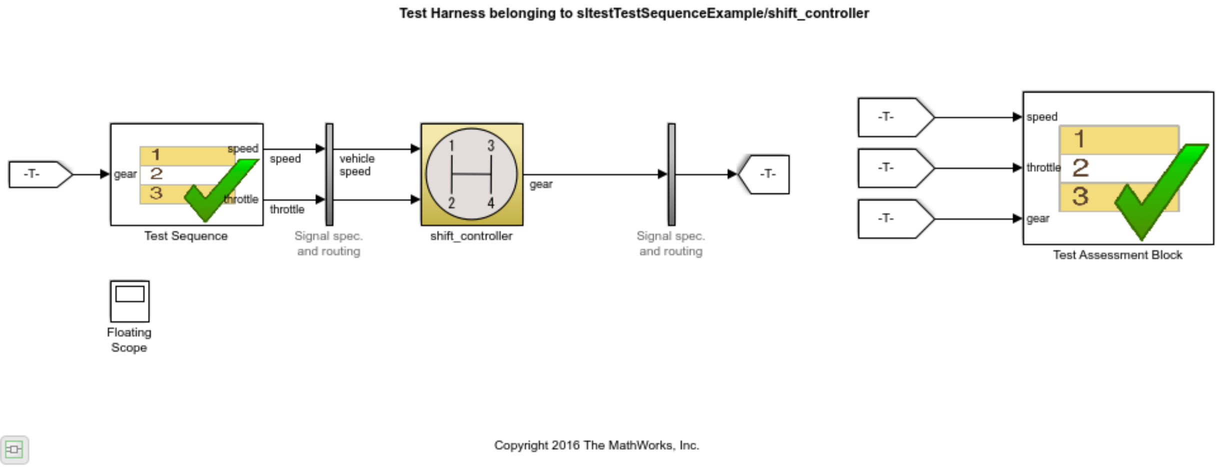


Testing Downshift Points of a Transmission Controller

This example shows how to create a Test Harness with a Test Sequence block as a source.

Copyright 2016 The MathWorks, Inc.

2. Click the badge on the `shift_controller` subsystem and open the `controller_harness` test harness.



Test Harness belonging to sltestTestSequenceExample/shift_controller

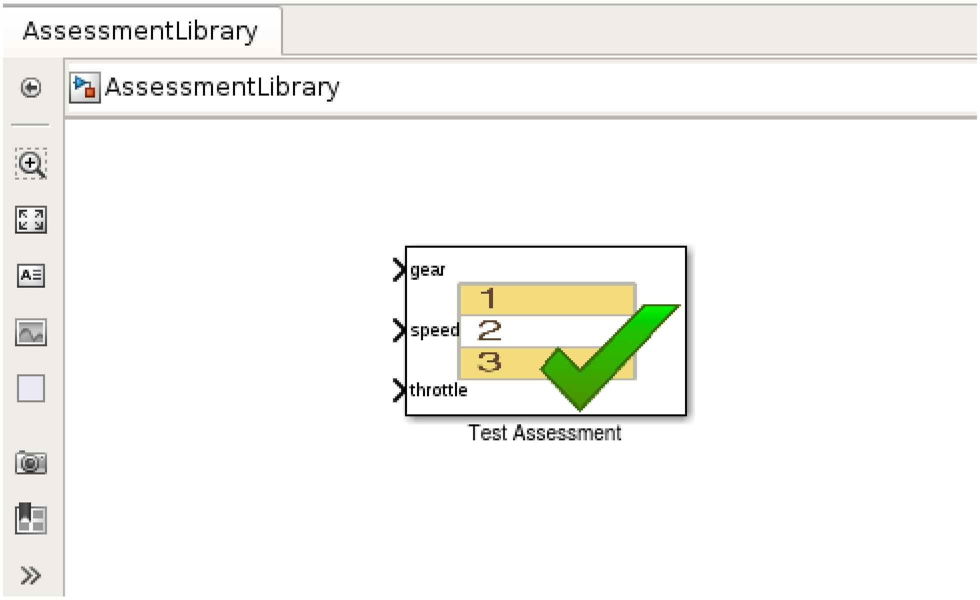
Copyright 2016 The MathWorks, Inc.

The Test Assessment block contains four assertions that define the assessment criteria:

```
assert(speed >= 0)
assert(throttle >= 0)
assert(throttle <= 100)
assert(gear > 0)
```

Create a Library for the Test Assessments

1. In the test harness, select **File > New > Library**.
2. Save the new library as `AssessmentLibrary` in a writable location on the MATLAB® path.
3. Copy the Test Assessment block from the test harness to the library, and then delete the Test Assessment block from the test harness.
4. Save the library.



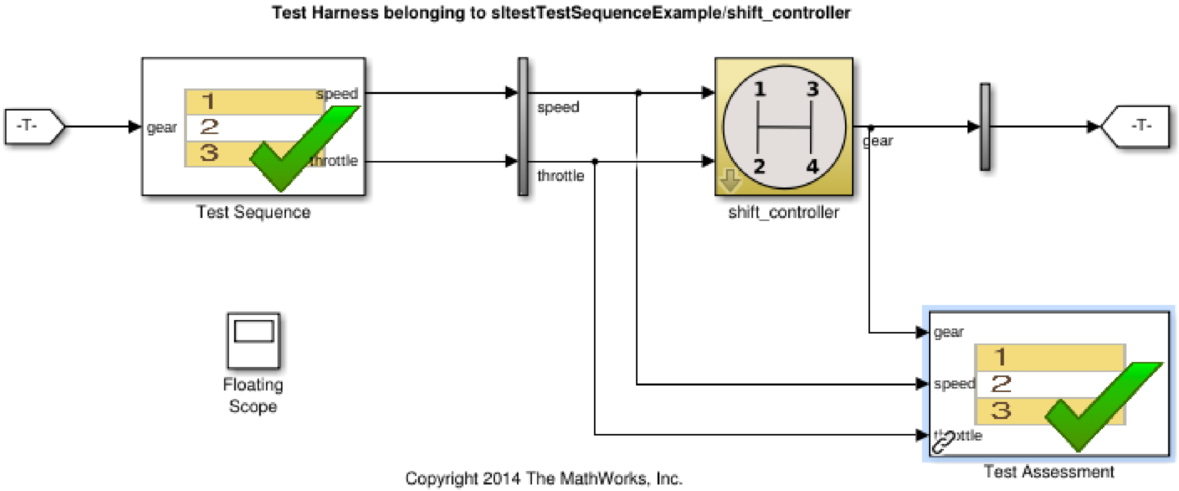
Create a Linked Test Assessment Block in Test Harnesses

Copy the Test Assessment block from the library to the test harness to create a linked block.

1. In the test harness, enable the library link display. Select **Display > Library Links > All**.

2 Copy the Test Assessment block from `AssessmentLibrary` into `controller_harness`. The block displays a library link badge.

3 Connect the signal inputs to the Test Assessment block.



Edit the Assessment Block in the Library

1 Unlock the library. Select **Diagram > Unlock Library**.

2 Add a fifth assertion to the Test Sequence block: `assert(gear < 5);`

3 Save and close the library. Closing locks the library.

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:

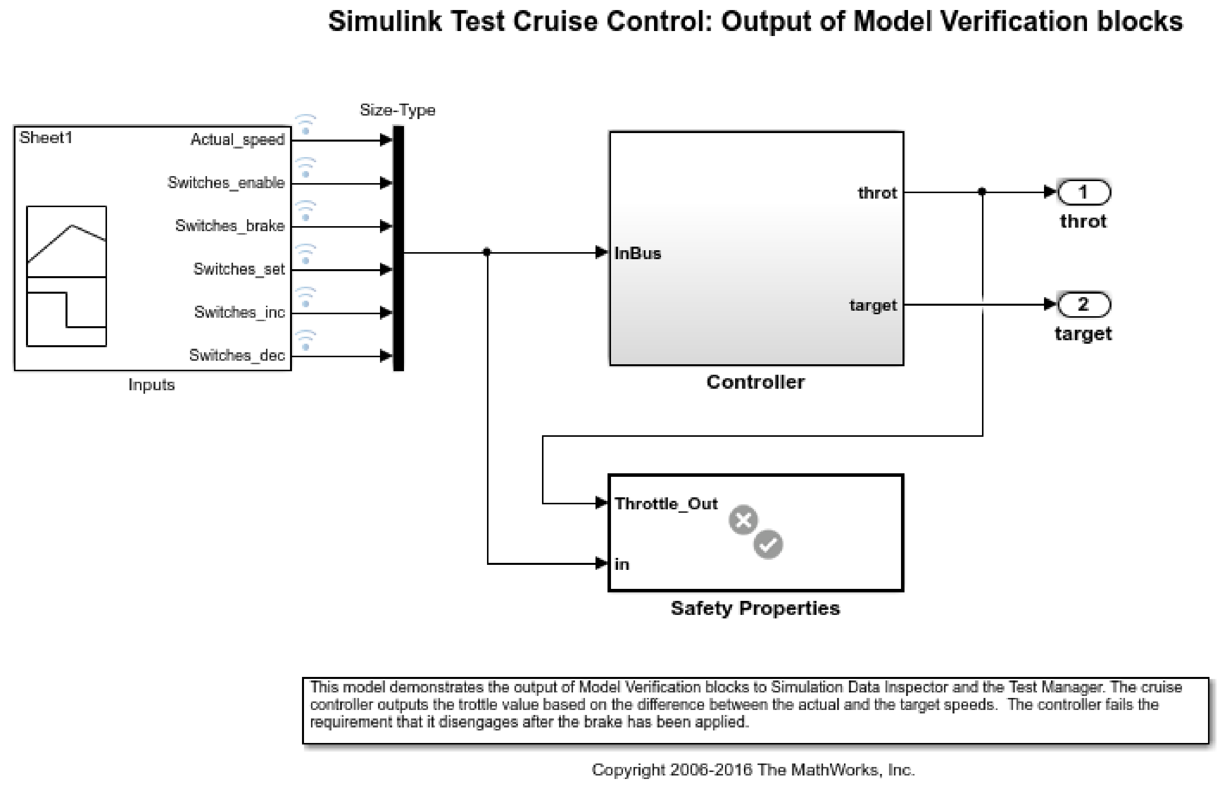
- Determine the time step when a failure occurs.
- Debug the model by comparing the verification result with relevant signals.
- Trace failures from the results to the model.

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.

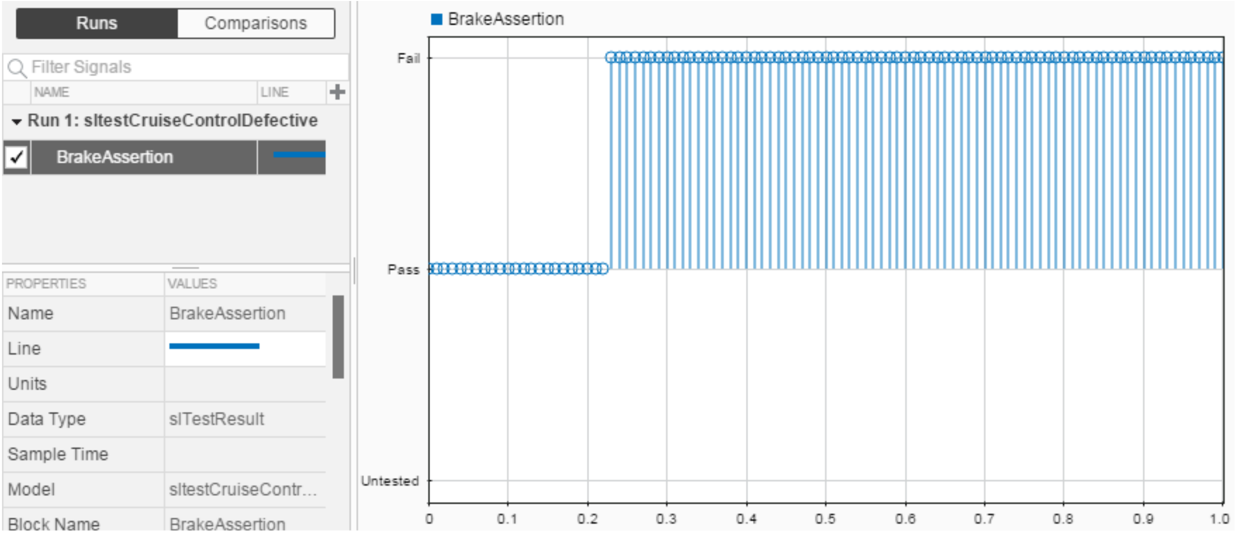
Copyright 2006-2016 The MathWorks, Inc.

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.

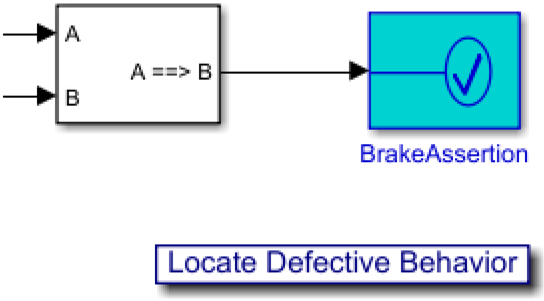
```
Simulink.sdi.view
```



PROPERTIES	VALUES
Name	BrakeAssertion
Line	▬
Units	
Data Type	slTestResult
Sample Time	
Model	sltestCruiseContr...
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.



Work with Message Viewer

The Message Viewer window has:

- A navigation toolbar, which contains:
  - The model hierarchy path
  - Toggle button to select an automatic or manual layout
  - Toggle button to chose to show or hide inactive lifelines
  - Buttons for saving and restoring information in the viewer, setting parameters on the block, and accessing the Message Viewer documentation
- A header pane, which contains the lifeline headers.
- A message pane, which displays the messages.

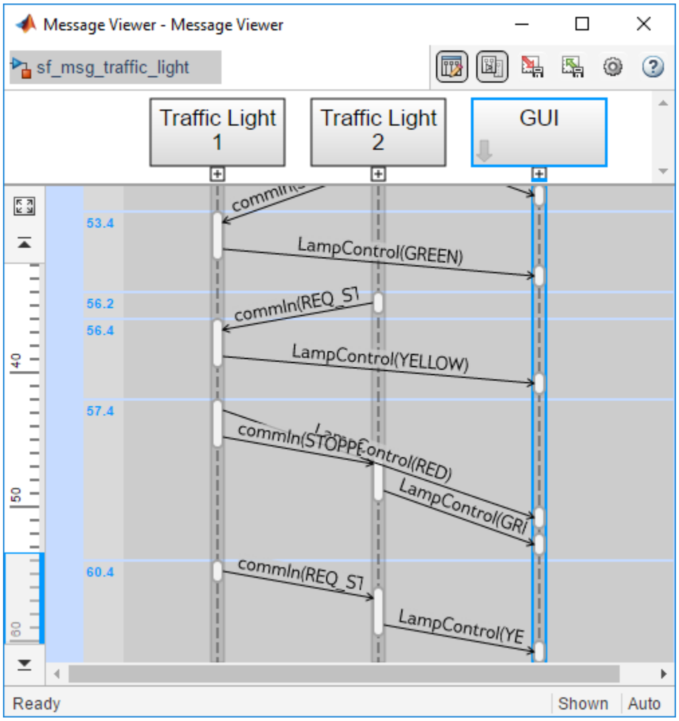
To see the interchange of messages between Stateflow charts during simulation, add a Message Viewer block to the model. You can visualize the movement of entities between blocks when simulating SimEvents® models. The Message Viewer block also displays function calls and calls from MATLAB Function blocks. For more information on function calls, see "Function Calls in Message Viewer" on page 3-73.

The Message Viewer block uses a Message Viewer window that acts like a sequence diagram showing how blocks interact using messages.

The Message Viewer enables you to view event data related to Stateflow chart execution and the exchange of messages between Stateflow charts. The Message Viewer shows where messages are created and sent, forwarded, received, and destroyed at different times during model execution. You can also view the movement of entities between SimEvents blocks. All SimEvents blocks that can store entities appear as lifelines on the Message Viewer. Entities moving between these blocks appear as lines with arrows. The Message Viewer also enables you to view calls to Simulink Function blocks and Stateflow MATLAB functions.

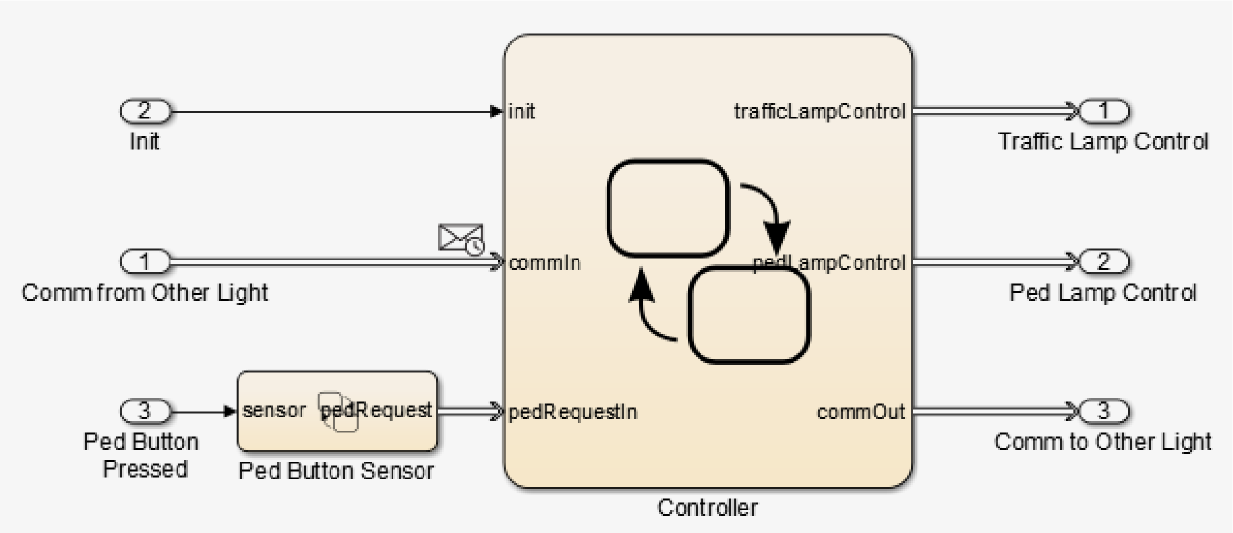
This topic uses the Stateflow example `sf_msg_traffic_light` to show you how to use the Message Viewer.

You can add one or more Message Viewer blocks to the top level of a model or any subsystem. If you place a Message Viewer block in a subsystem that does not have messages, the Message Viewer informs you that no messages are available to display. A viewer can be inactive if, for example, it is in a subsystem that has been commented out. In such a case, the Message Viewer displays that it is inactive.



Visualize Messages

Consider this subsystem, Traffic Light1:



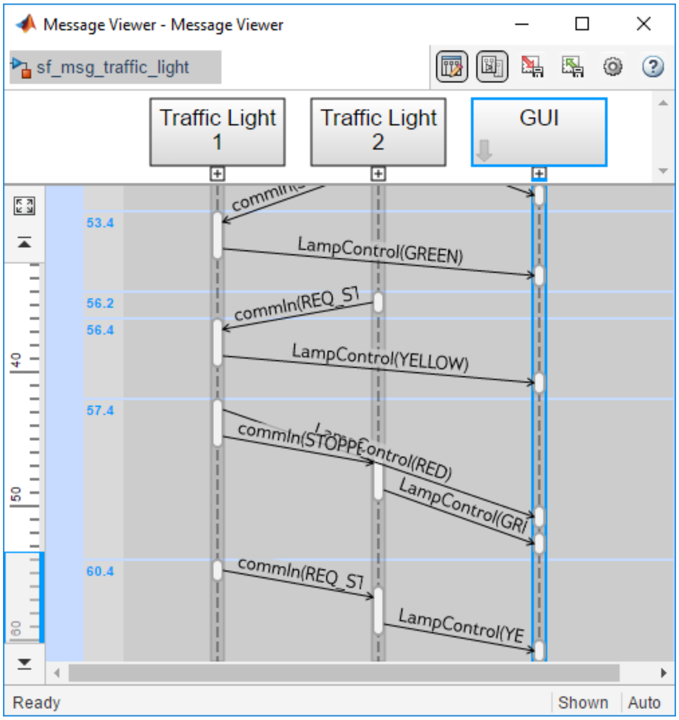
Traffic Light1 contains two Stateflow charts.

- Controller
- Ped Button Sensor

The charts in this subsystems use messages to exchange data. As messages pass through the system, you can view them in a Message Viewer.

Add a Message Viewer block from the Stateflow library to a subsystem or model whose messages you want to see. When you open a Message Viewer block and simulate the model:

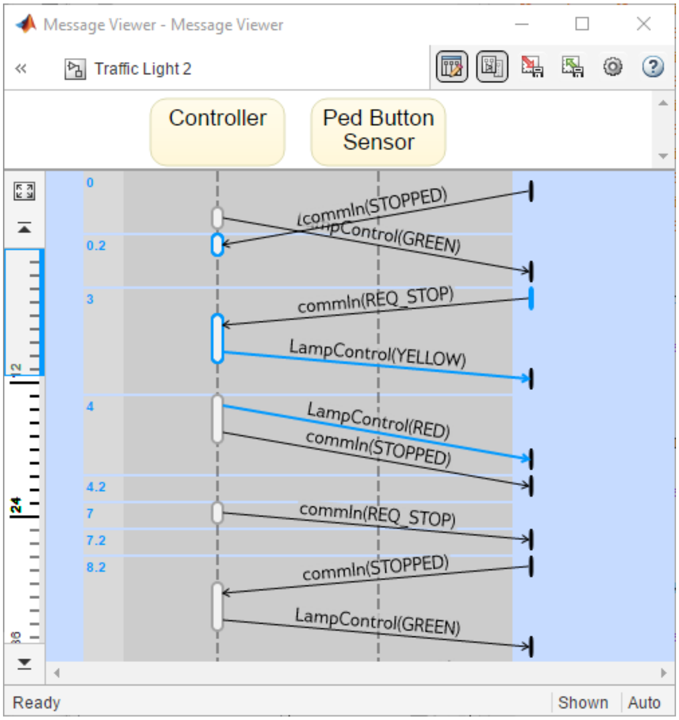
1 Observe the contents of the Message Viewer.



The header (top) pane of a Message Viewer shows the lifeline headers. In this example, the lifelines are the two Traffic Light blocks and the GUI. Lifeline headers show the name of the corresponding blocks in the model that generate or act on messages. The top of the lifeline is a header, which corresponds to a block in the model. Gray headers with straight edges correspond to subsystems. Yellow headers with rounded edges correspond to Stateflow charts. In the header pane, the lifeline hierarchy corresponds to the model hierarchy. When the model is paused or stopped, you can expand and close lifelines.

In the message pane, a thick gray lifeline indicates that you can expand the lifeline to see the children in the lifeline. Clicking a lifeline name opens the corresponding block in the model. Messages between lifelines display in the message pane. Message lines are arrows from the sender to the receiver. For more information on navigation in the message page, see “Navigation in Message Viewers” on page 3-72.

2 To show the children of a lifeline, click the expander under a parent lifeline ⊞.

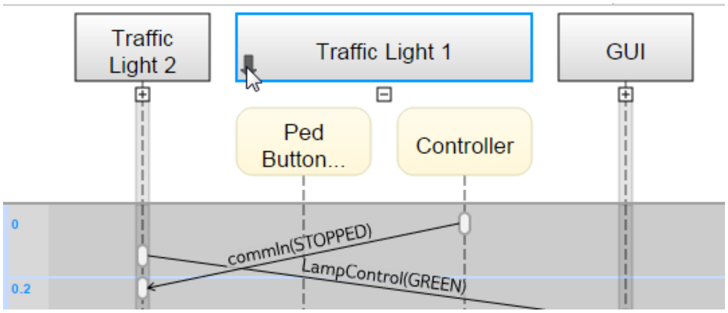


3 Hide the children lifelines by clicking the **Lifeline Stack**

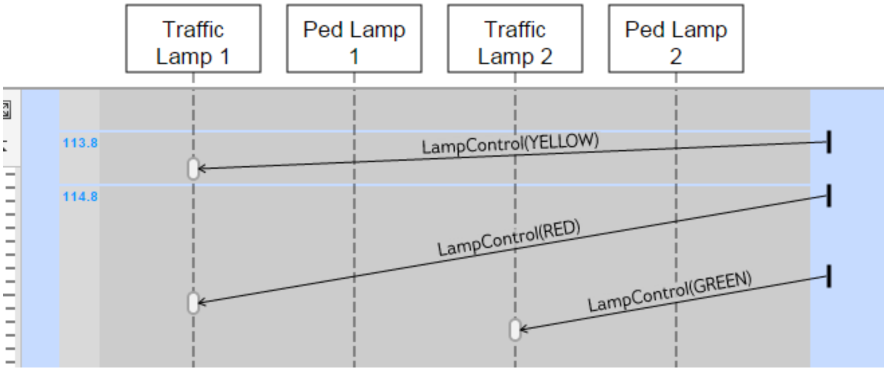
«

.

4 Make a lifeline the root of focus for the viewer. Hover over the bottom left corner of the lifeline header and click the arrow. Alternatively, use the navigation toolbar at the top of a Message Viewer. A Message Viewer displays the current root lifeline path and shows its child lifelines.

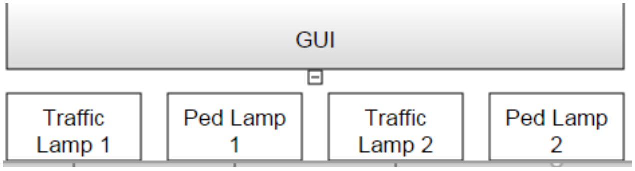


Any external sending and receiving events display in the diagram gutter ▌. To highlight the associated block in the model, click the gutter.

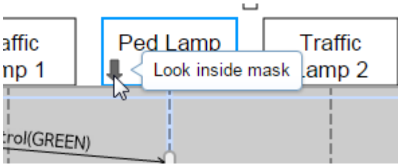


You can use the navigation toolbar to move the current root up and down the lifeline hierarchy. To move up the current root one level, hit the **Esc** key.

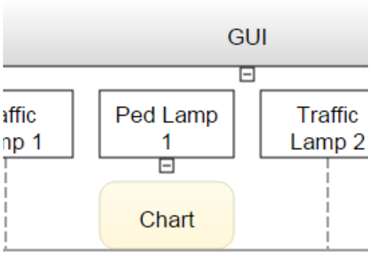
This graphic also illustrates how the Message Viewer displays masked subsystems. The Traffic Lamp 1, Ped Lamp 1, Traffic Lamp 2, and PED Lamp 2 are masked subsystems. The Message Viewer displays masked subsystems as white blocks.



5 To show the children of a masked subsystem, hover over the bottom left corner of the masked and subsystem and click the arrow.



The child lifeline displays.

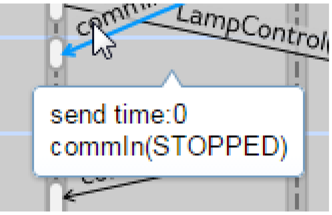


6 Activations that correspond to executions of the lifeline are at the start and end of each message line.

If a message line is not completely shown, hover over the line. You can also, hover over a truncated message label to see it in its entirety. In this example, the send time of the **commIn** message line is not visible. To see it, hover over the message line.

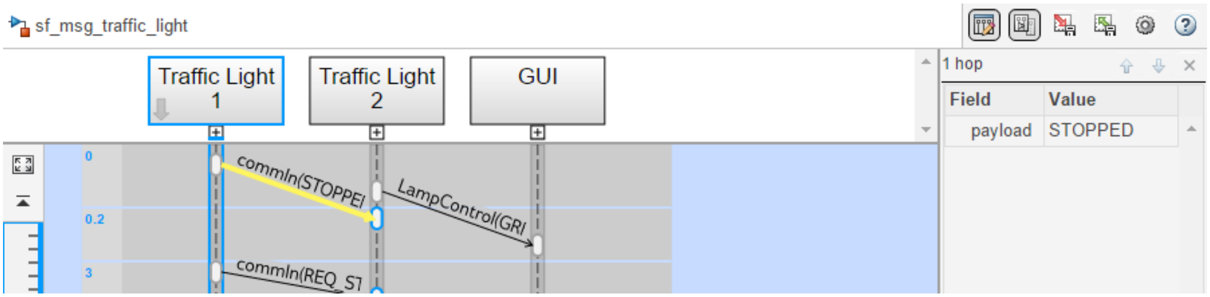
If you hover over an activation that represents a function call, the function prototype is displayed in the tool tip.

If you hover over partially shown activation symbols, the times for any truncated activations also appear.



send time:0
commIn(STOPPED)

7 A Message Viewer shows the interactions (hops) that a message or function call goes through in its lifetime. It also shows message and function call payloads. To highlight the hops for a message and display its payload, click the corresponding message line. See the result in the *payload inspector* to the right. Use **Search Up** and **Down** buttons to move through the hops.



Field	Value
payload	STOPPED

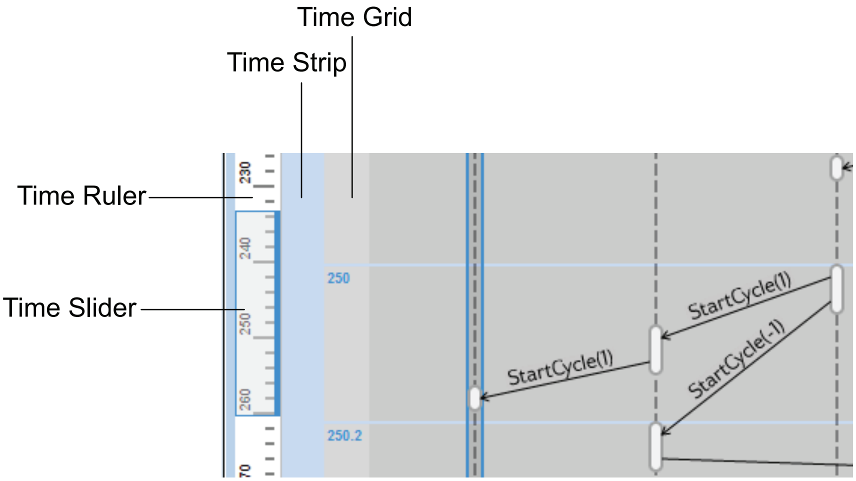
Redisplay of Information in Message Viewer

A Message Viewer block saves the order and states of lifelines between simulation runs. Similarly, when you close and reopen a Message Viewer, it preserves the last open lifeline state. To save a particular viewer state with the block, click in the navigation toolbar. Saving the model saves the state information across sessions. Use to load the saved settings.

Time in Message Viewers

A Message Viewer shows message events vertically, ordered in time. Multiple events in Simulink can happen at the same time. Conversely, there can be long periods of times during simulation with no events. As a consequence, time in the message pane is nonlinear. Each time grid row, bordered by two blue lines, contains events that occur at the same simulation time. The time strip gives the times of the events in that grid row.

The time ruler shows linear simulation time. To show messages in that simulation time range, use the scroll wheel or drag the time slider up and down the time ruler.



- To navigate to the beginning and end of the simulation, click the **Go to first event** and **Go to last event** buttons.
- To zoom the ruler, hold the space bar and use the mouse wheel. This action increases and decreases the amount of time ruler space the slider occupies.
- The time ruler covers the whole simulation time. To see the entire simulation duration on the time ruler, click the **Fit to view** button ⛶.
- To reset the zoom to 100%, hold **Ctrl** + **0**.

Navigation in Message Viewers

To scroll in the header and message panes, use the mouse wheel. In addition,

- The header pane has a vertical scroll bar.
- The message page has a horizontal scroll bar at the bottom that scrolls both panes.

To pan in the message pane, move the mouse while holding down either the middle mouse button or space bar. This action moves both panes.

You can scale the view in two ways:

- Fit all lifeline headers to window — Press the space bar.
- Zoom by a fixed increment to a predefined minimum or maximum value — Press **Ctrl-** or **Ctrl+**. Alternatively, hold the space bar and use the mouse wheel.

Zooming does not scale the navigation toolbar or time ruler.

Function Calls in Message Viewer

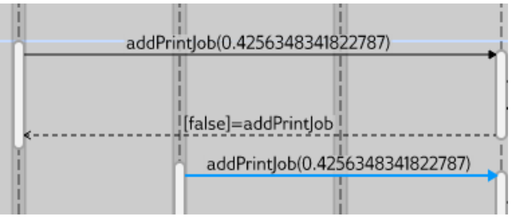
The Message Viewer block displays these function calls and replies to them.

Function Call Type	Support
Calls to Simulink Function blocks	Fully supported
Calls to Stateflow graphical or Stateflow MATLAB functions	• Scoped — Select the **Export chart level functions** chart option. Use the `chartName.functionName` dot notation. • Global — Select the **Treat exported functions as globally visible** chart option. You do not need the dot notation.

The Message Viewer block does not display these function calls:

- Function calls connected to function-call subsystems.

For an example of functions calls in Message Viewer, see `slexPrinterExample`. The Message Viewer displays function calls with solid lines terminated with solid arrows and a label with the format *`function_name`*`(`*`argument_list`*`)`. Replies to function calls display as dashed lines with open arrows and a label with the format `[`*`argument_list`*`]=`*`function_name`*.


addPrintJob(0.4256348341822787)
[false]=addPrintJob
addPrintJob(0.4256348341822787)

See Also

Message Viewer | Message Viewer | Message Viewer

More About

- “How Messages Work in Stateflow Charts” (Stateflow)

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.

For information about running multiple simulations with unchanged generated code, see "Prevent Code Changes in Multiple Simulations" (Embedded Coder).

Also see "Code Generation of Subsystems" (Simulink Coder) in the Simulink Coder™ documentation.

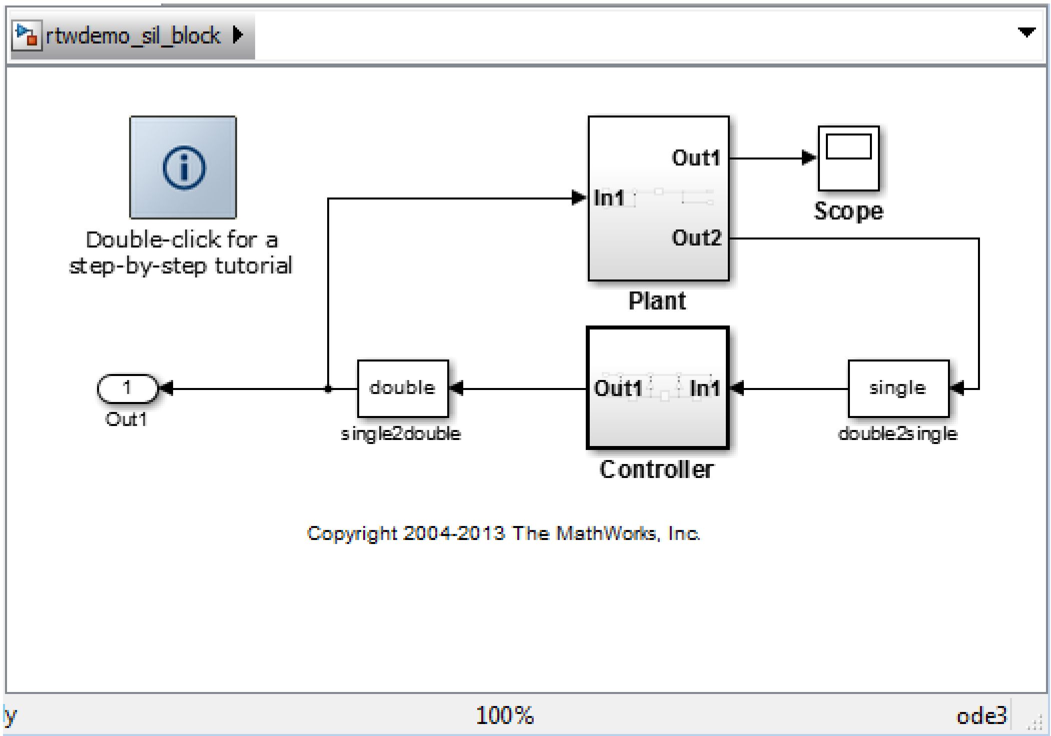
The example models a closed-loop controller-plant system. The controller regulates the plant output.

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

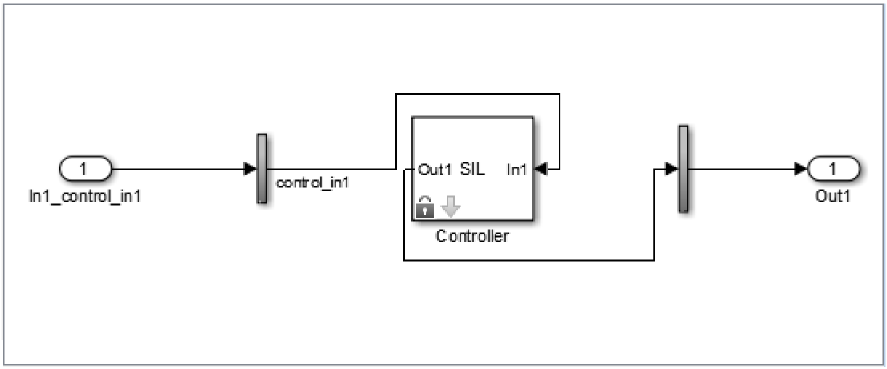
```
set_param(bdroot,'SignalLogging','on','SignalLoggingName',...
'SIL_signals','SignalLoggingSaveFormat','Dataset')
```

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`
   - **Initial harness configuration**: `Verification`
   - **Verification Mode**: `Software-in-the-loop (SIL)`
   - Select **Open harness after creation**

   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.

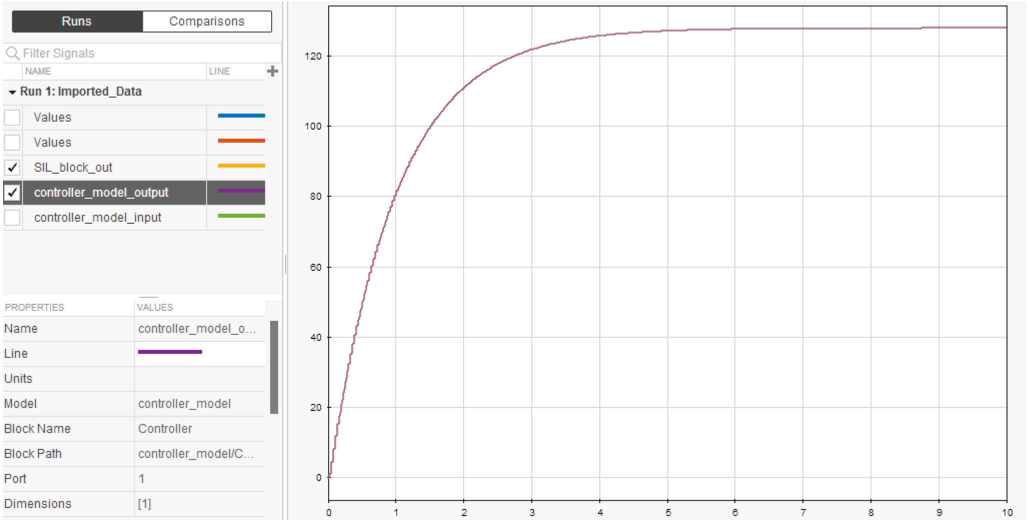
   - Set **Import from** to: `Base workspace`.

- 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) in the Simulink documentation.



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.

Test Code in S-Functions

In this section...

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.

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 is set up to work only on a 64–bit Windows® platform.

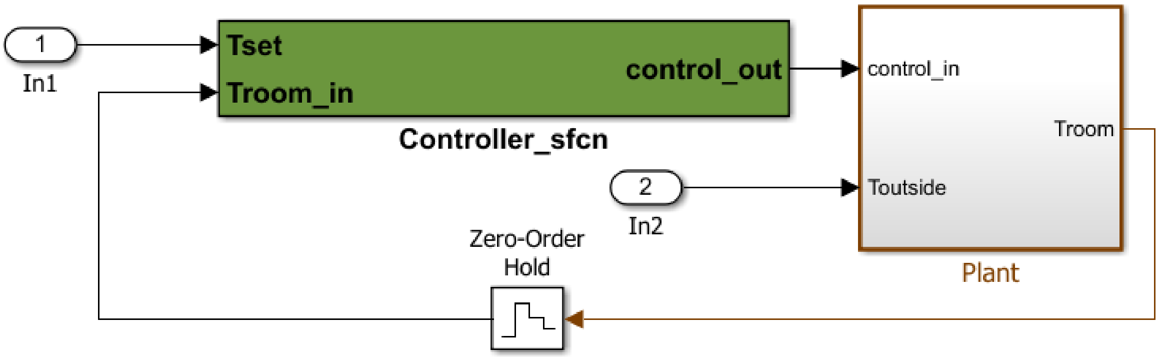
Set Up the Working Environment

1. Navigate to the model.

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

   - `sltestHeatpumpSfunExample.slx`
   - `sltestHeatpumpBusPostLoadFcn.mat`
   - `PumpDirection.m`
3. Open the model.

   ```
   open_system('sltestHeatpumpSfunExample')
   ```



Copyright 1990-2016 The MathWorks, Inc.

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
`|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

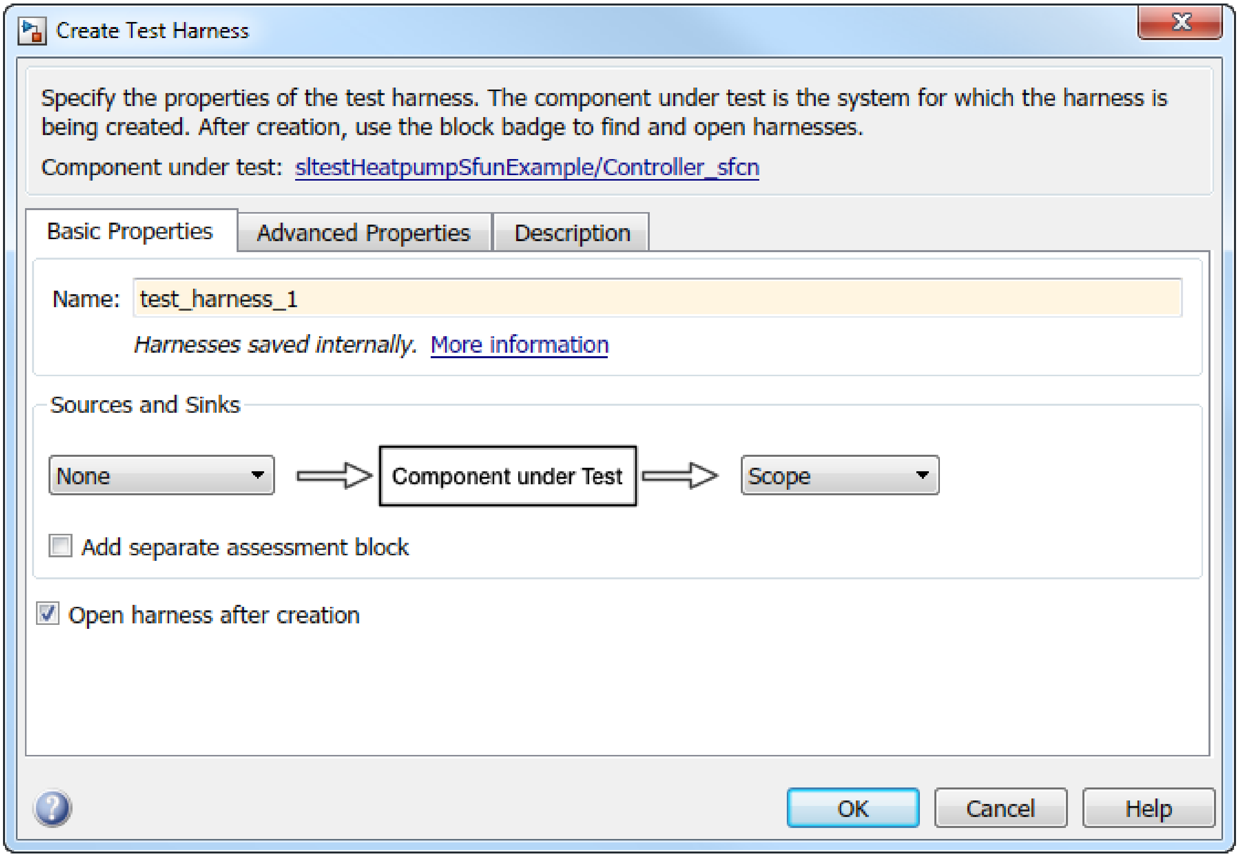
Create a Test Harness for the Controller

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

2 Set the harness properties.

   In the **Basic Properties** tab:

   - Set **Name** to `test_harness_1`
   - Set **Sources and Sinks** to **None** and **Scope**



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: sltestHeatpumpSfunExample/Controller_sfcn

Name: test_harness_1

Harnesses saved internally. More information

Sources and Sinks: None → Component under Test → Scope

□ Add separate assessment block

☑ Open harness after creation

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

Add Inputs and Set Simulation Parameters

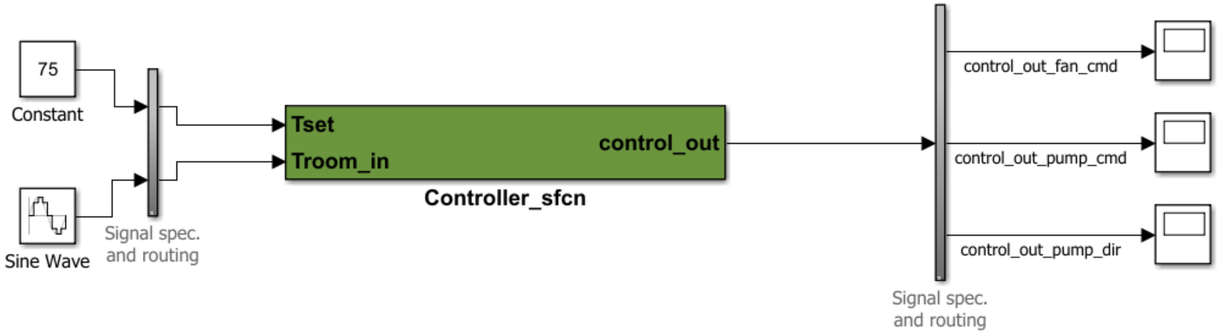
1 Create a test input for the harness with a constant Tset and a time-varying Troom_in. Connect a Constant block to the Tset input and set the value to 75.

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_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.

Create a Test Case and Obtain a Baseline

1 In the test harness, select all three output signals from the Output Conversion Subsystem and right-click one of them. Select **Log Selected Signals**.

2 Open the Test Manager. Select **Analysis** > **Test Manager**.

3 From the Test Manager toolstrip, click **New** to create a test file. Name and save the test file.

4 In the test case, under **System Under Test** , click the button to load the current model into the test case.

5 Expand **Test Harness** and select `test_harness_1` .

6 Under **Baseline Criteria**, click **Capture** to record a baseline data set from the model specified under **System Under Test**. Save the baseline data set to the working folder. The model runs and the baseline criteria appear in the table.

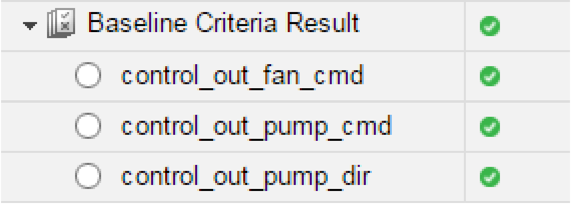
Run the Test Case and View Results

1 Run the selected test case.

The Test Manager switches to the **Results and Artifacts** pane, and the new test result appears at the top of the table.

2 Expand the results until you see the baseline criteria result.

The overall baseline test passes.



▾ Baseline Criteria Result	✓
○ control_out_fan_cmd	✓
○ control_out_pump_cmd	✓
○ control_out_pump_dir	✓

Simulink Test Manager Introduction

Introduction to the Test Manager

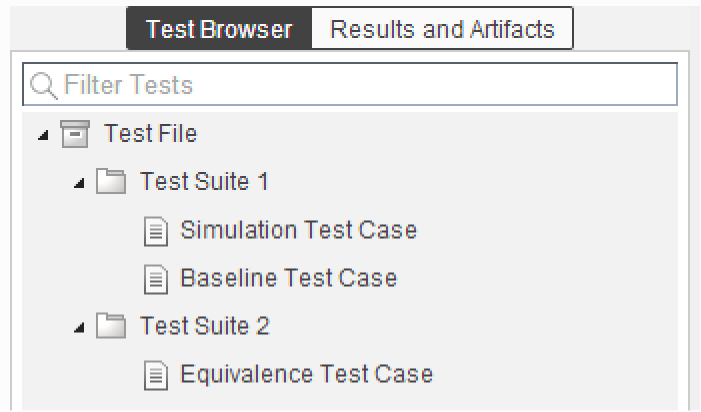
In this section...

Test Manager Description

The Test Manager in Simulink Test enables you to automate Simulink model testing and organize large sets of tests. A model test is performed using test cases where criteria are specified to determine a pass-fail outcome. The test cases are run from the Test Manager. At the end of a test, the test case results are organized and viewed in the Test Manager.

Test Creation and Hierarchy

Test cases are contained within a hierarchy of test files and test suites in the **Test Browser** pane of the Test Manager. A test file can contain multiple test suites, and test suites can contain multiple test cases.



There are three types of test case templates to choose from in the Test Manager. Each test case uses a different set of criteria to determine the outcome of a test.

- **Baseline**: compares signal outputs of a simulation to a baseline set of signals. The comparison of the simulation output and the baseline must be within the absolute or relative tolerances to pass the test, which is defined in the **Baseline Criteria** section of the test case.
- **Equivalence**: compares signal outputs between two simulations. The comparison of outputs must be within the absolute or relative tolerances to pass the test, which is defined in the **Equivalence Criteria** section of the test case.
- **Simulation**: checks that a simulation runs without errors, which includes model assertions.

Test Results

Results of a test are given using a pass-fail outcome. If all of the criteria defined in a test case is satisfied, then a test passes. If any of the criteria are not satisfied, then the test fails. Once the test has finished running, the results are viewed in the **Results and Artifacts** pane. Each test result has a summary page that highlights the outcome of the test: passed, failed, or incomplete. The simulation output of a model is also shown in the results section. Signal data from the simulation output can be visually inspected using the Simulation Data Inspector.

Share Results

Once you have completed the test execution and analyzed the results, you can share the test results with others or archive them. If you want to share the results to be viewed later in the Test Manager, then you can export the results to a file. To archive the results in a document, you can generate a report, which can include the test outcome, test summary, and any criteria used for test comparisons.

Related Examples

- “Test Model Output Against a Baseline” on page 6-9
-
-

Test Manager Test Cases

Manage Test File Dependencies

In this section...
"Package a Test File Using Simulink Projects" on page 6-2
"Find Test File Dependencies and Impact" on page 6-4
"Share a Test File with Dependencies" on page 6-8

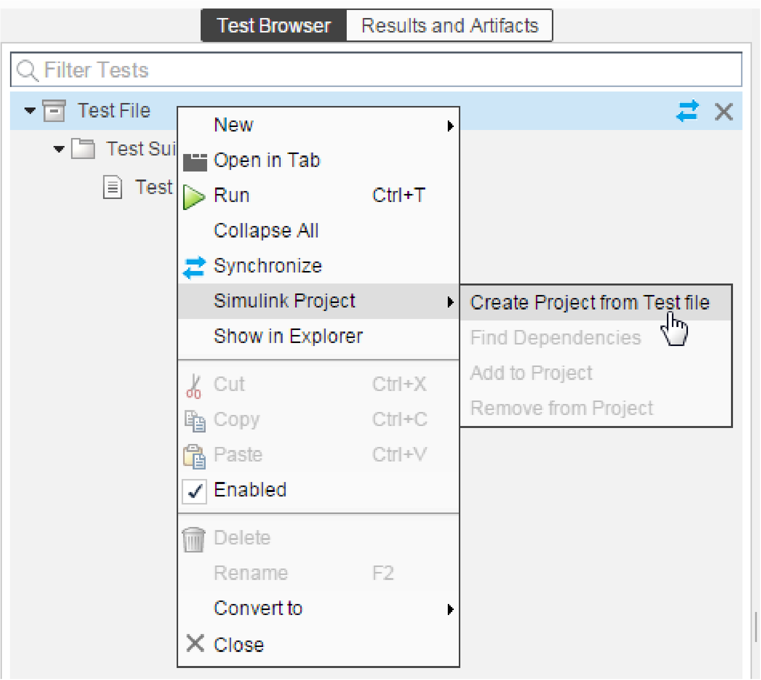
A test file can be simple and contain only a few test cases. For such a test file, the file dependencies for models, test requirements, input files, callbacks, and baseline data can be manageable. When test files become large and complex, it is difficult to track and manage file dependencies. You can use Simulink projects to help manage these dependencies. Projects are especially helpful if you want to package and share a test file.

Package a Test File Using Simulink Projects

1 In the **Test Browser**, right-click the test file.

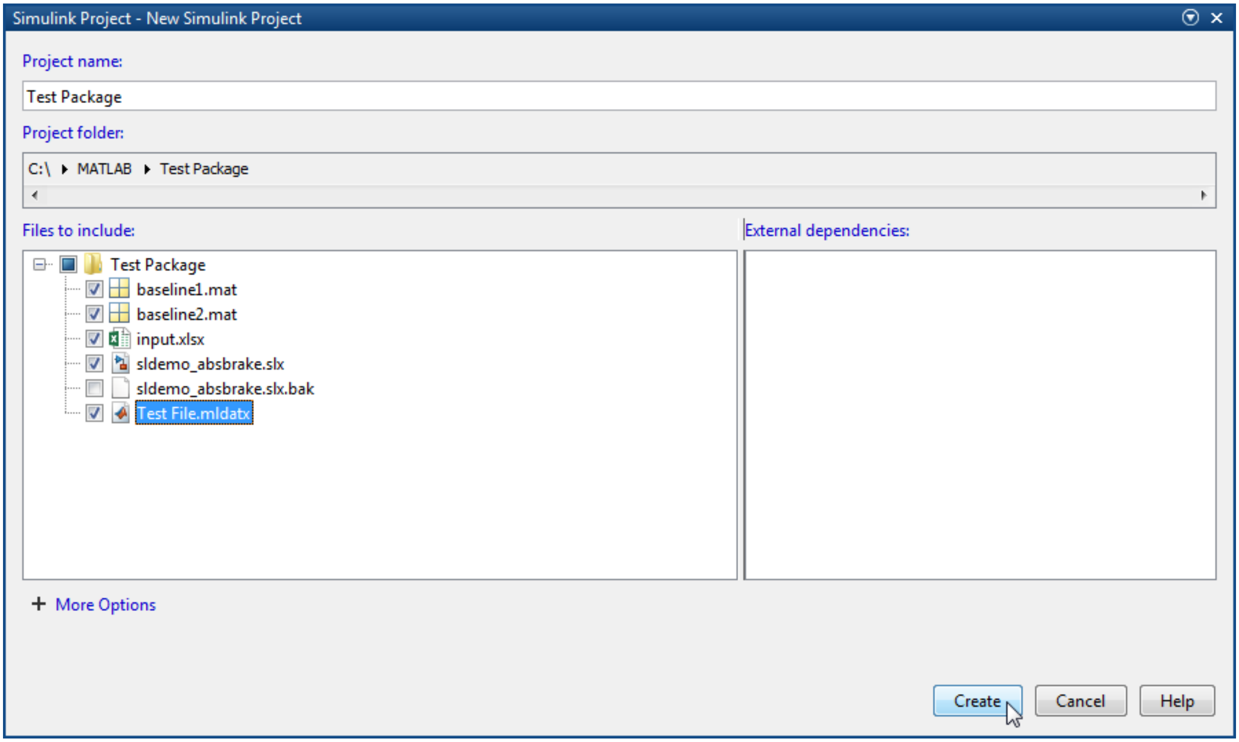
2 Select **Simulink Project** > **Create Project from Test File**.

   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:

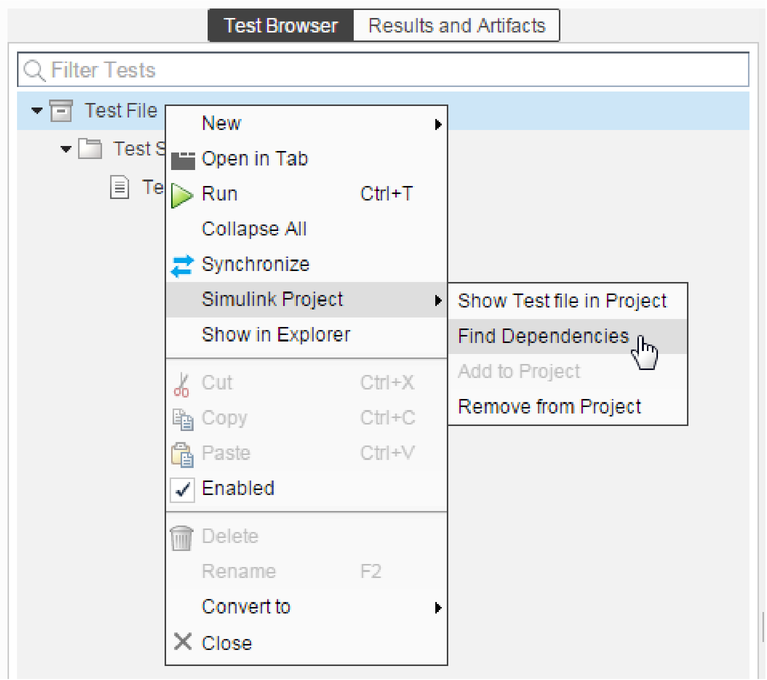
- ⊟ Test Package
  - ☑ baseline1.mat
  - ☑ baseline2.mat
  - ☑ input.xlsx
  - ☑ sldemo_absbrake.slx
  - ☐ sldemo_absbrake.slx.bak
  - ☑ Test File.mldatx

External dependencies:

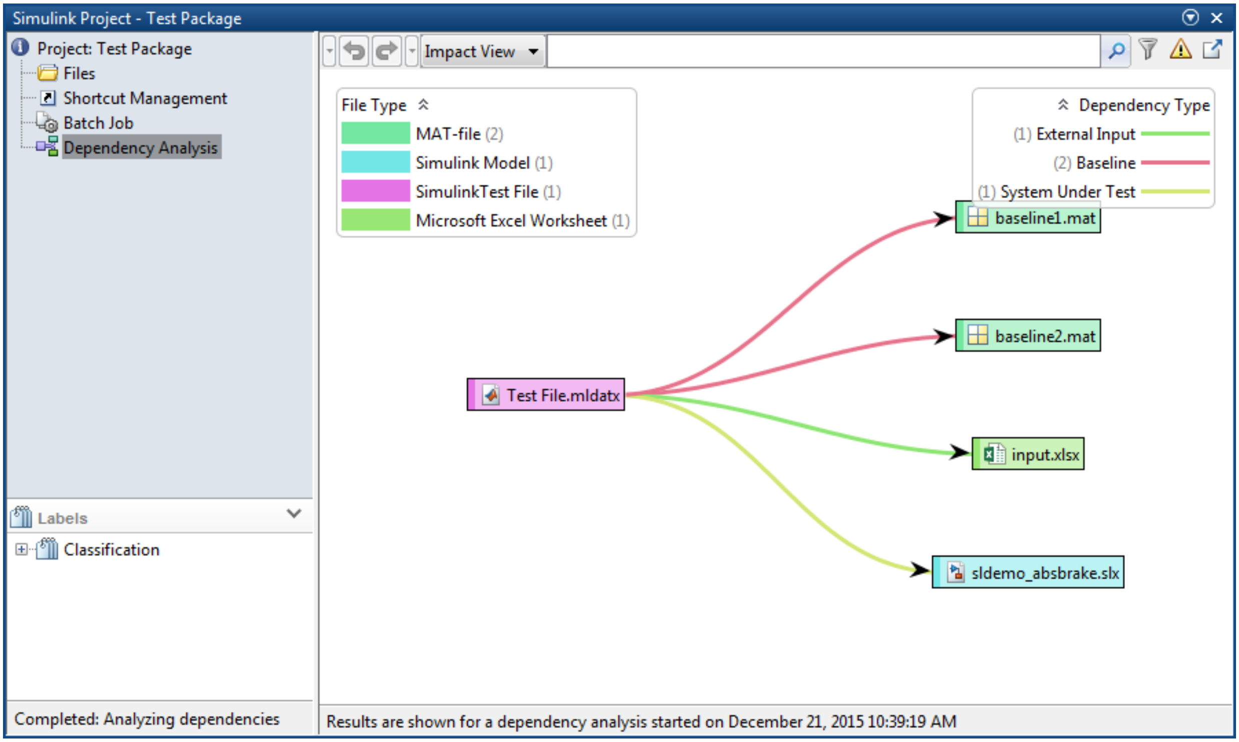
Find Test File Dependencies and Impact

If you have a test file saved in a Simulink project, then you can find the file dependencies.

1 Right-click the test file. Select **Simulink Project** > **Find Dependencies**.



Simulink Projects shows a graph of file dependencies.

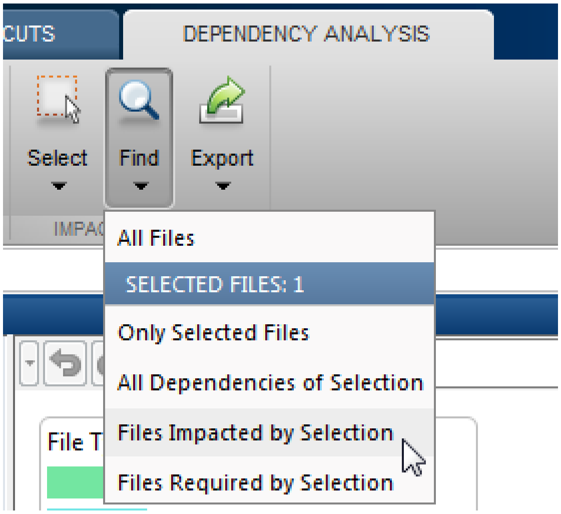


Completed: Analyzing dependencies

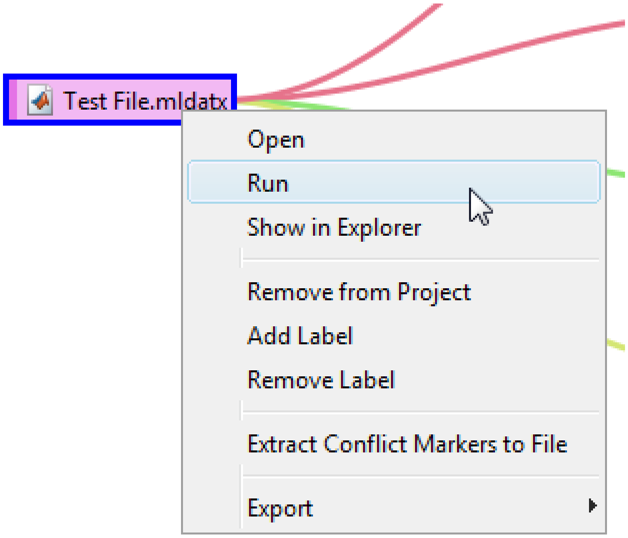
Results are shown for a dependency analysis started on December 21, 2015 10:39:19 AM

If you want to change a model or requirement, then you can find the impact that the change could have on testing.

1. In the dependency graph, select the item that would want to assess the impact for.
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.

Related Examples

- “What Are Simulink Projects?” (Simulink)

Test Model Output Against a Baseline

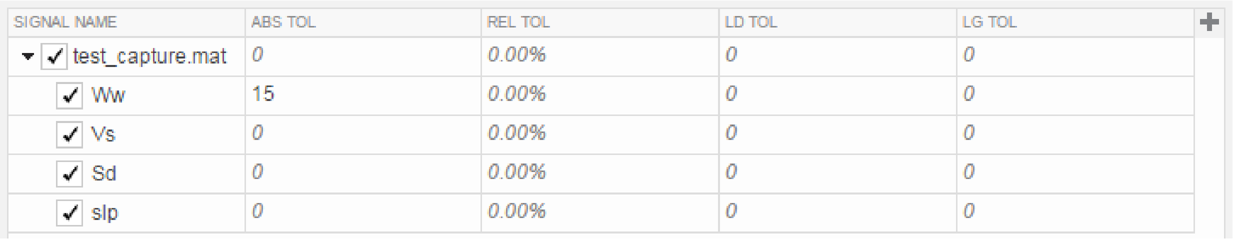
To test the simulation output of a model against a defined baseline, use a baseline test case. In this example, use the `sldemo_absbrake` model to compare the simulation output to a baseline that is captured from an earlier state of the model.

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 new 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. Under the **Baseline Criteria** section, click **Capture** to record a baseline from the model specified under **System Under Test**.

   Save the baseline to a location. After you save the baseline MAT-file, the model runs and the baseline criteria appear in the table.
7. Expand the **Baseline Criteria** section. Set the **Absolute Tolerance** of the `Ww` signal to `15`.



SIGNAL NAME	ABS TOL	REL TOL	LD TOL	LG TOL
☑ test_capture.mat	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

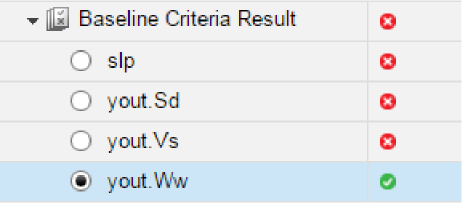
To add or remove columns in the baseline criteria table, click the column selector button +. For more information about tolerances and criteria, see “Apply Tolerances to Test Criteria” on page 6-51.

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** to run the selected test case.

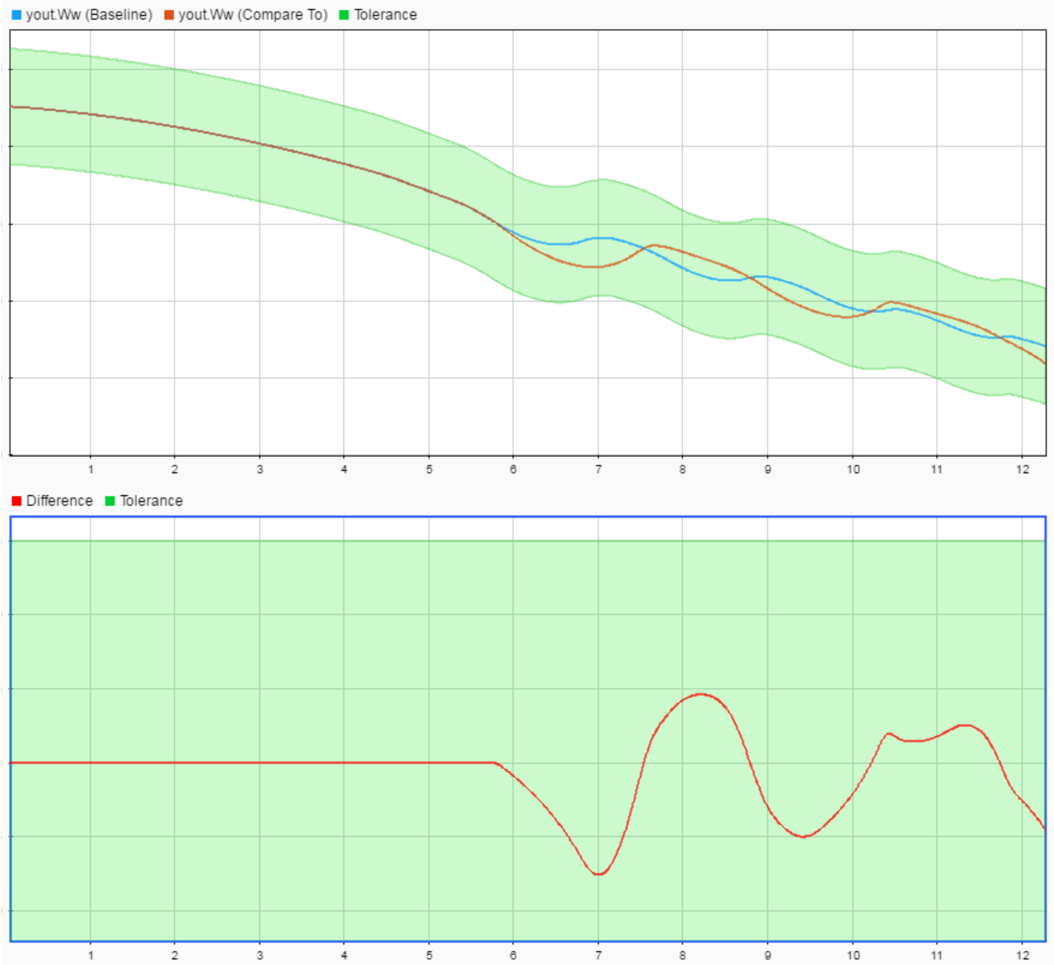
   The Test Manager switches to the **Results and Artifacts** pane, and the new test result appears at the top of the table.
4. Expand the results until you see the baseline criteria result. Right-click the result and select **Expand All Under**.

   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.

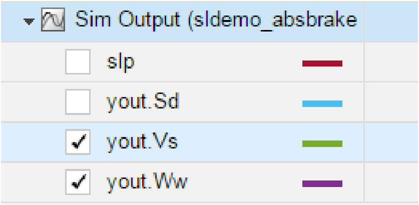
   

Baseline Criteria Result	✗
○ slp	✗
○ yout.Sd	✗
○ yout.Vs	✗
● yout.Ww	✓

   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

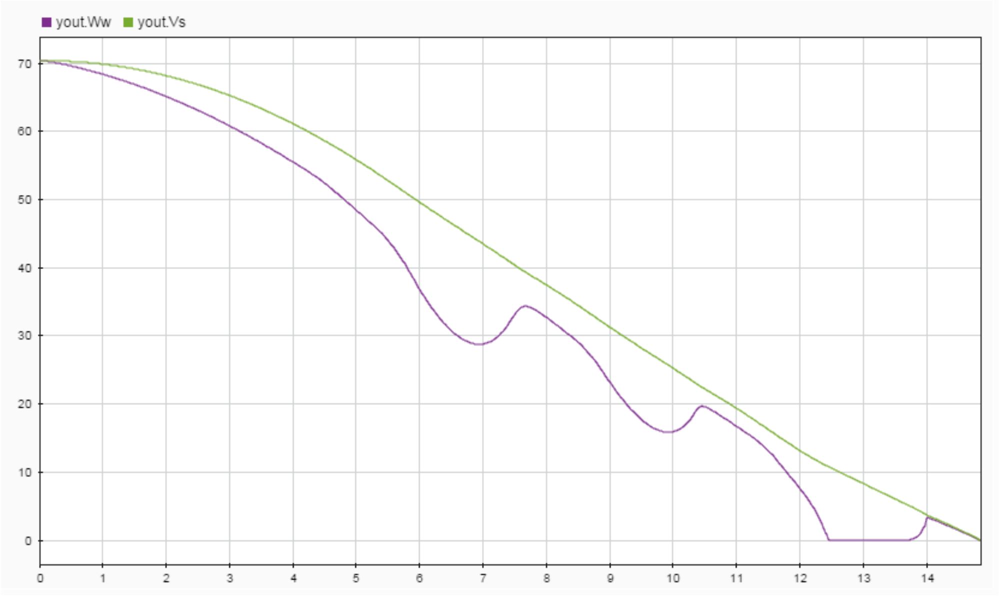
□ slp

□ 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 7-9.

Related Examples

- “Apply Tolerances to Test Criteria” on page 6-51

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. The pass-fail criteria used for a simulation test case is that the simulation finishes without any errors.

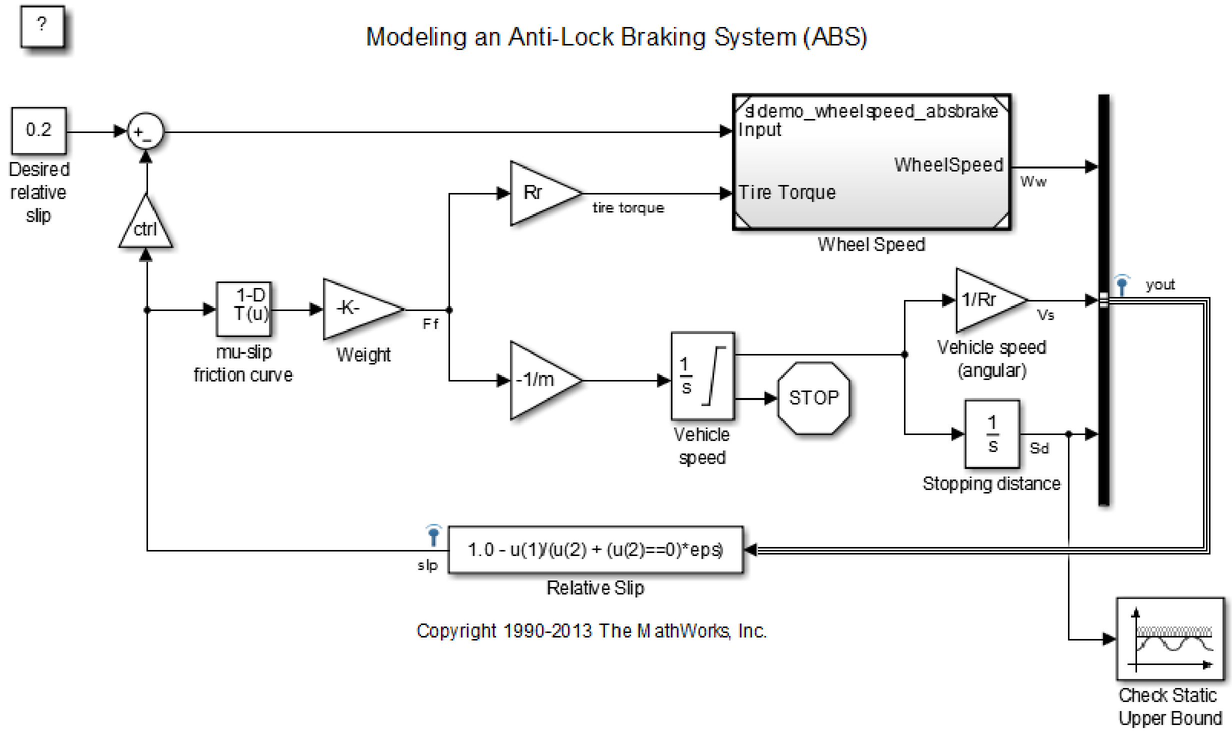
Configure the Model

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`.

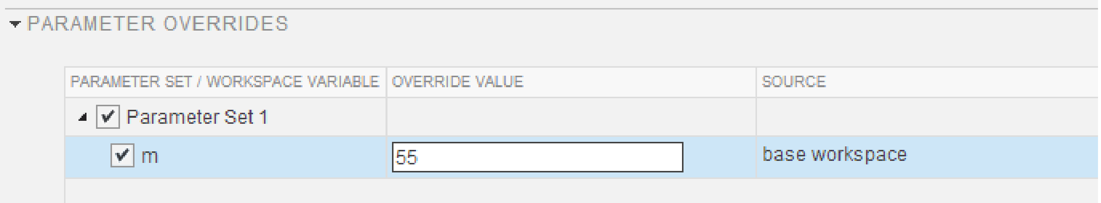
5 In the test case, under **System Under Test**, click the **Use current model** button to assign the `sldemo_absbrake` model to the test case.

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 OVERRIDES

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.

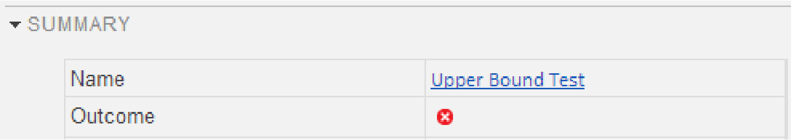
2 In the Test Manager toolstrip, click **Run**. The test results appear in the **Results and Artifacts** pane.

View Test Results

1 Expand the test results, and double-click `Upper Bound Test`.

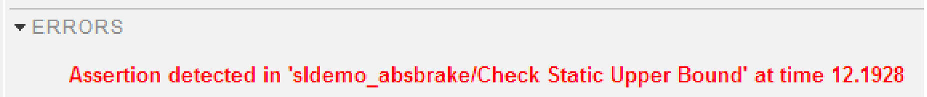
A new tab displays the outcome and results summary of the simulation test.

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.



SUMMARY

Name	Upper Bound Test
Outcome	⊗



ERRORS

Assertion detected in 'sldemo_absbrake/Check Static Upper Bound' at time 12.1928

Generate Test Cases from Model Components

In this section...

The Test Manager can generate a list of test cases for you based on the components in your model. Test cases can be generated from:

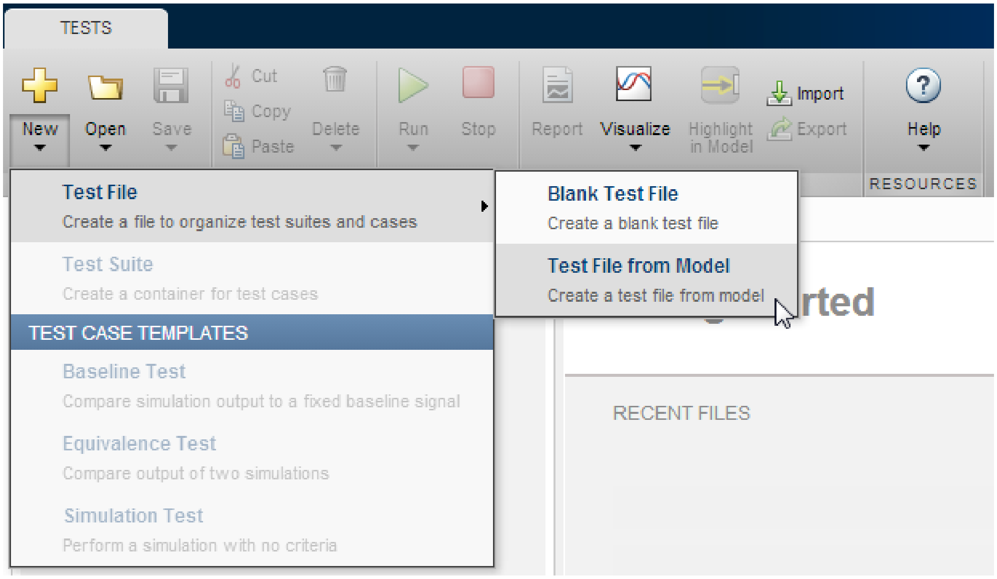
- Signal Builder block in the top model
- Test harnesses from the top model or any subsystem
- Signal Builder block at the top level of a test harness

If there are multiple Signal Builder blocks in the top model, then the Test Manager does not create any test cases from Signal Builder blocks.

If you want to generate a test case for a model subsystem using a harness, see "Generate Test for a Subsystem" on page 6-20.

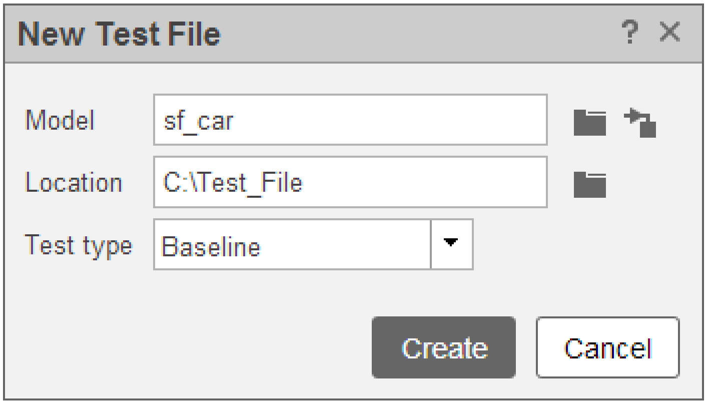
Generate the Test Cases

1. In the Test Manager, click the **New** arrow and select **Test File** > **Test File from Model**.



2 In the **New Test File** dialog box, select the model and location. The model must be on the MATLAB path.

3 Select the **Test type** to generate for the test cases.



Model: sf_car

Location: C:\Test_File

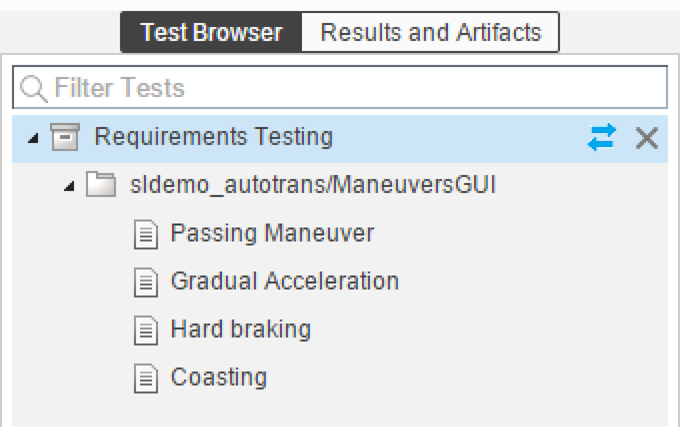
Test type: Baseline

4 Click **Create**.

Synchronize Test Cases

If you add components to your model, such as Signal Builder groups or test harnesses, you can synchronize tests by automatically creating test cases in the Test Manager. Also, if you remove model components, then you can disable or delete test cases in the Test Manager when you synchronize. In the Test Manager **Test Browser** pane, you can synchronize your model and test file using the synchronization button ⇄ next to the test file name.

For example, the `sldemo_autotrans` model has a Signal Builder block with four groups by default. If you automatically create test cases from the model using **New** > **Test File** > **Test File from Model**, then test cases are created using the Signal Builder groups.



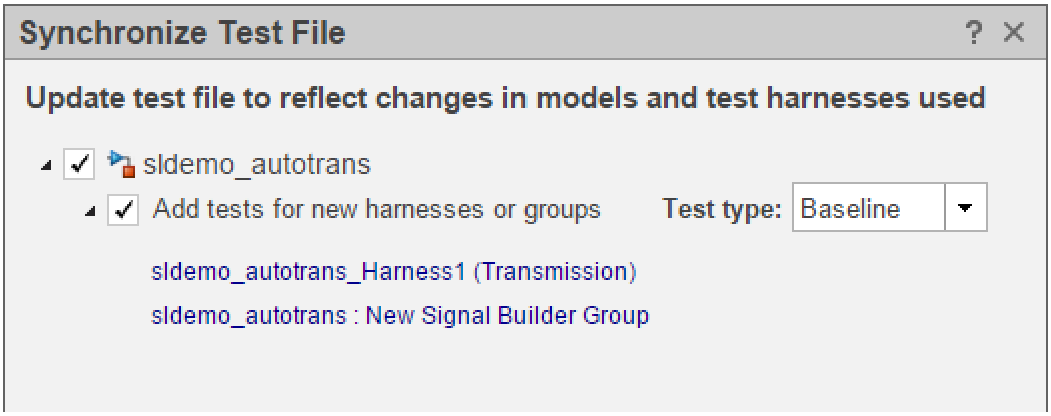
Requirements Testing ⇄ ✕
sldemo_autotrans/ManeuversGUI
Passing Maneuver
Gradual Acceleration
Hard braking
Coasting

If you add another Signal Builder group, `New Signal Builder Group`, and a test harness, `sldemo_autotrans_Harness1`, then you can add test cases for these model components. Synchronize the model and test file.

1 In the Test Manager, hover over the test file name that you want to synchronize.

2 Click the synchronization button ⇄ next to the test file name.

3 In the synchronization dialog box, add or remove any test cases, and select the test case type.



Synchronize Test File

Update test file to reflect changes in models and test harnesses used

☑ sldemo_autotrans

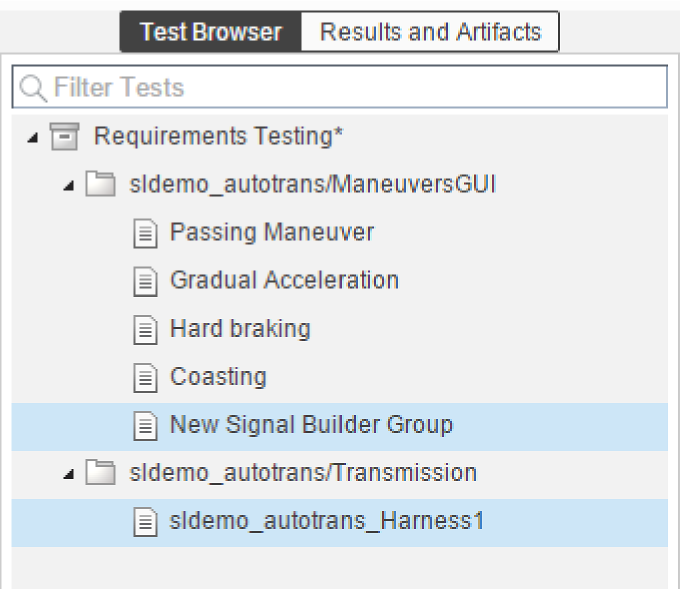
☑ Add tests for new harnesses or groups   Test type: Baseline

sldemo_autotrans_Harness1 (Transmission)

sldemo_autotrans : New Signal Builder Group

4 To complete the synchronization, click **Update Test File**.

In the **Test Browser** pane, the new test cases appear in the test file.



Requirements Testing*

sldemo_autotrans/ManeuversGUI

Passing Maneuver

Gradual Acceleration

Hard braking

Coasting

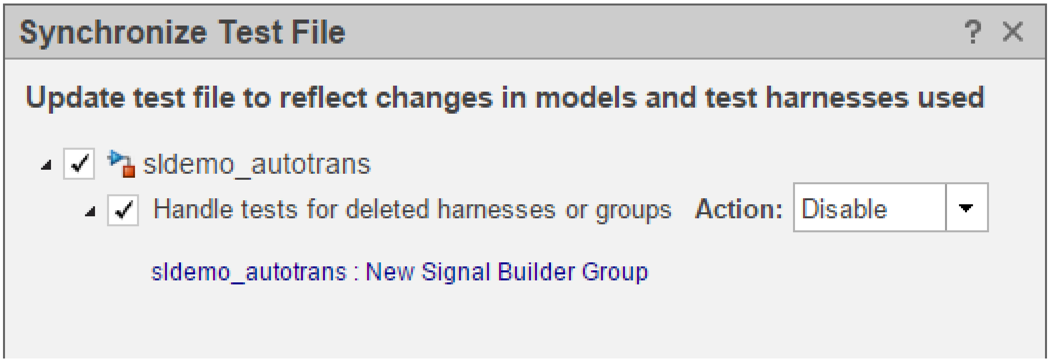
New Signal Builder Group

sldemo_autotrans/Transmission

sldemo_autotrans_Harness1

If you remove model components and synchronize the test file, then you can remove or disable a test case using the **Action** menu. For example, if you remove `New Signal`

`Builder Group` from the model, then the synchronization dialog box shows the deleted Signal Builder group.



Synchronize Test File

Update test file to reflect changes in models and test harnesses used

☑ sldemo_autotrans

☑ Handle tests for deleted harnesses or groups **Action:** Disable

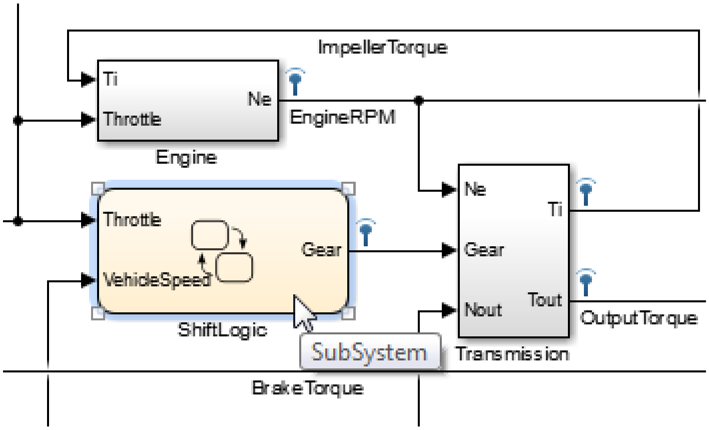
sldemo_autotrans : New Signal Builder Group

Generate Test for a Subsystem

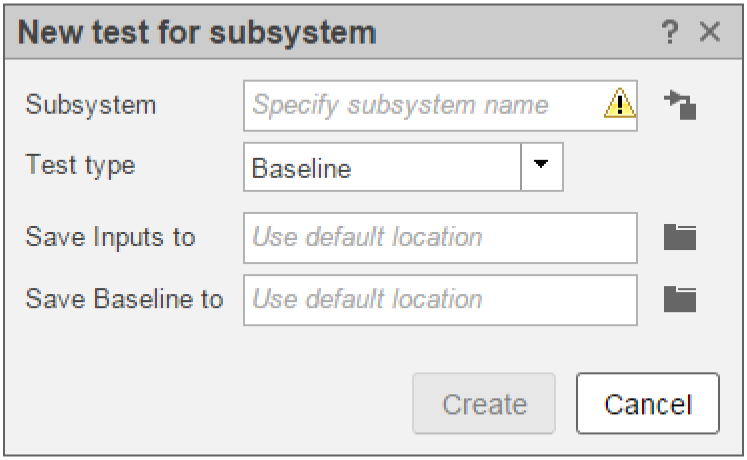
If you want to isolate a subsystem and test it on its own, then you can create a test in the Test Manager. If you create a test from a subsystem from the Test Manager, the Test Manager creates a test case and a harness for the subsystem. It then assigns the harness to the system under test. Finally, it simulates the model, records the inputs for the harness, and assigns input and baseline data files to the test case.

To create a test from a subsystem:

1. Open a model that you want to create the test for the subsystem. In this example, the model is `sldemo_autotrans`.
2. In the model, select the subsystem you want to test.



3 In the Test Manager, from the toolstrip, click the **New** arrow and select **Test for Subsystem**. A dialog box opens to help you create the test.


New test for subsystem

Subsystem: *Specify subsystem name*

Test type: Baseline

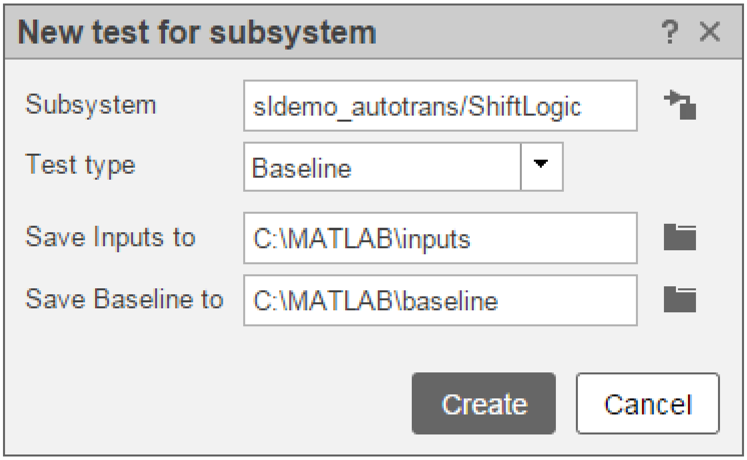
Save Inputs to: *Use default location*

Save Baseline to: *Use default location*

4 To use the selected subsystem in the model, in this case `ShiftLogic`, click the **Use currently selected subsystem** button .

5 Select the test case type you want to use for this subsystem.

6 Specify the file name and path of the inputs file and baseline file, if applicable.



New test for subsystem

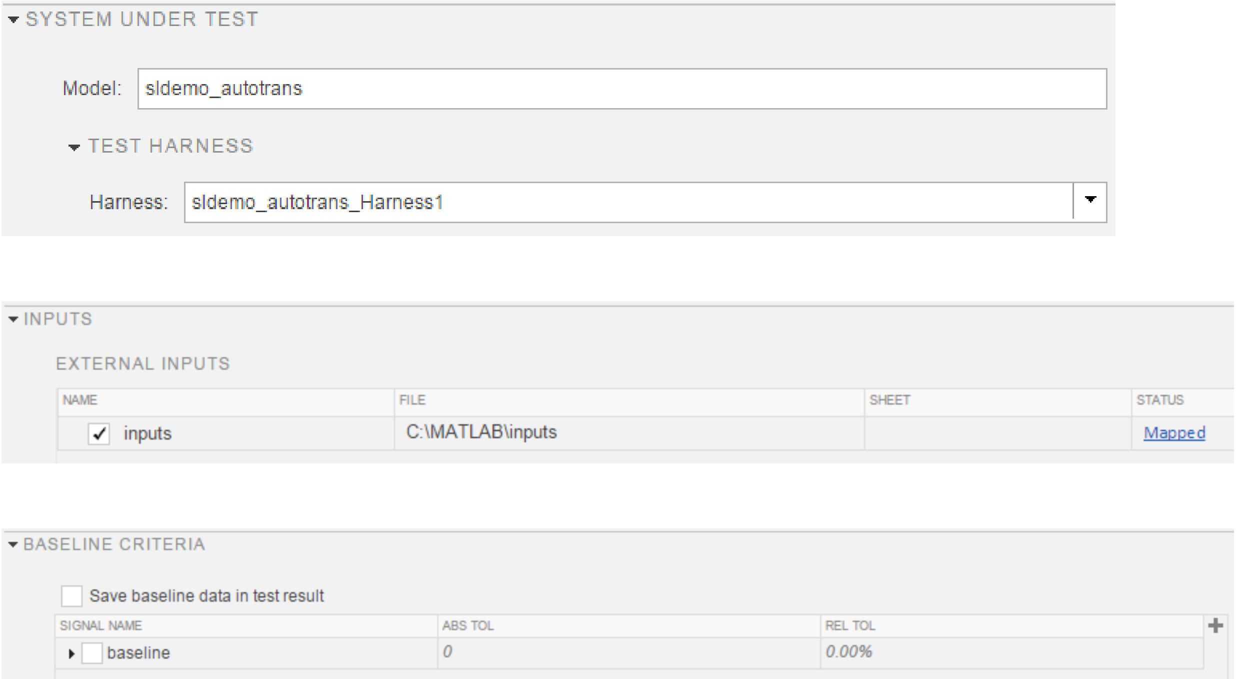
Subsystem: sldemo_autotrans/ShiftLogic

Test type: Baseline

Save Inputs to: C:\MATLAB\inputs

Save Baseline to: C:\MATLAB\baseline

7 Click **Create**. The Test Manager creates a test harness, logs the signals in the model for the inputs, and simulates the model. When simulation is done, the inputs and baseline sections of the test case are created for you.



SYSTEM UNDER TEST

Model: sldemo_autotrans

TEST HARNESS

Harness: sldemo_autotrans_Harness1

INPUTS

EXTERNAL INPUTS

NAME	FILE	SHEET	STATUS
☑ inputs	C:\MATLAB\inputs		Mapped

BASELINE CRITERIA

□ Save baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL
□ baseline	0	0.00%

The **Test for Subsystem** feature has some limitations on logging certain Simulink semantics and data types. The following are not supported:

- Function call
- State
- If-action
- Physical
- Merge
- Variable-size data type

Use External Inputs in Test Cases

In this section...
"Use MAT-File for Inputs" on page 6-24
"Use Microsoft Excel File for Inputs" on page 6-24

If you have external model inputs from MAT-files or Microsoft® Excel file sheets, then you can use these inputs in a test case. Map external inputs to the model in the **Inputs** section. You can import multiple external input files to a test case, but you can select only one external input set to execute when the test runs.

For more information about input mapping, supported data types or formats, and mapping results, see "Map Root Inport Signal Data" (Simulink).

Use MAT-File for Inputs

To add a MAT-file as an external input:

1. Expand the **Inputs** section in the test case.
2. Under the **External Inputs** table, click **Add**.
3. Specify a MAT-file.
4. Under **Input Mapping**, choose a mapping mode. For more information about mapping modes, see "Map Root Inport Signal Data" (Simulink).
5. Click **Map Inputs**. The **Mapping Status** table shows the port and signal mapping.

   For more information about troubleshooting the mapping status, see "Understand Mapping Results" (Simulink).
6. Click **Apply**.

Use Microsoft Excel File for Inputs

The Root Inport Mapper tool supports Microsoft Excel spreadsheets only for Windows systems. For Microsoft Excel spreadsheets:

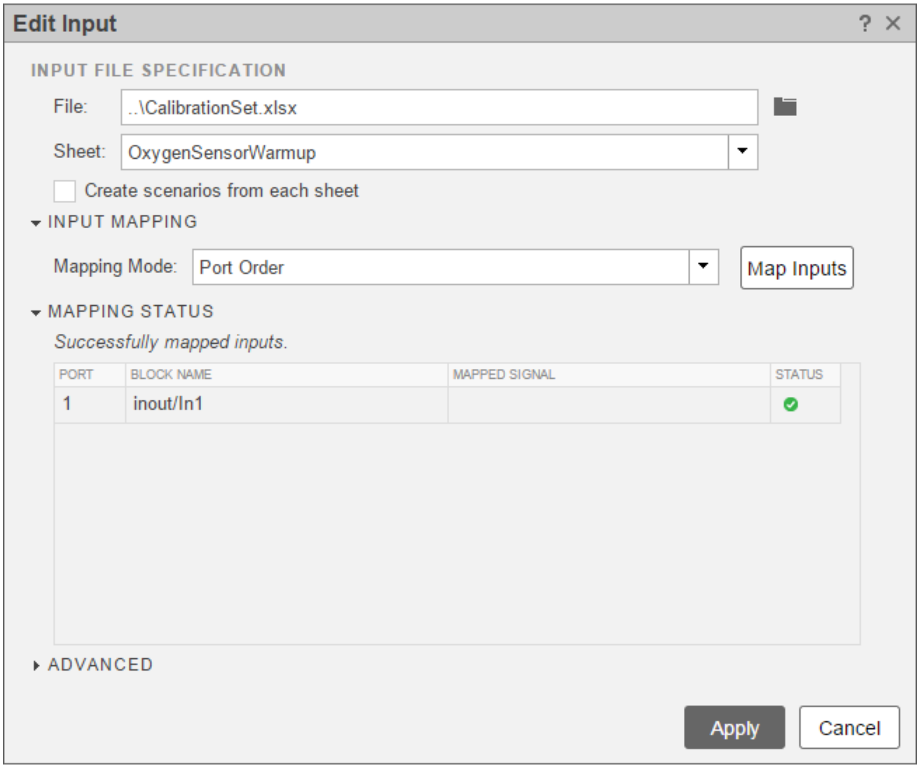
- The tool interprets each worksheet as a Simulink.SimulationData.Dataset data set.
- Each worksheet name must be a valid MATLAB variable name.

- The tool interprets the first row of a worksheet as signal names. If you do not specify a signal name, the tool assigns a default one using the format `Signal#`.
- If no columns have signal names, the tool assigns signal names using the format `Signal#,` where # increments with each additional signal.
- Signal-name columns must be filled in. If there are empty signals, the tool returns an error at import.
- The tool interprets the first column as time. In this column, the time values must increase.
- The tool interprets the remaining columns as signals.

To add a Microsoft Excel file as an external input:

1 Expand the **Inputs** section in the test case.

2 Under the **External Inputs** table, click **Add**.



Edit Input

INPUT FILE SPECIFICATION

File: ..\CalibrationSet.xlsx

Sheet: OxygenSensorWarmup

□ Create scenarios from each sheet

INPUT MAPPING

Mapping Mode: Port Order

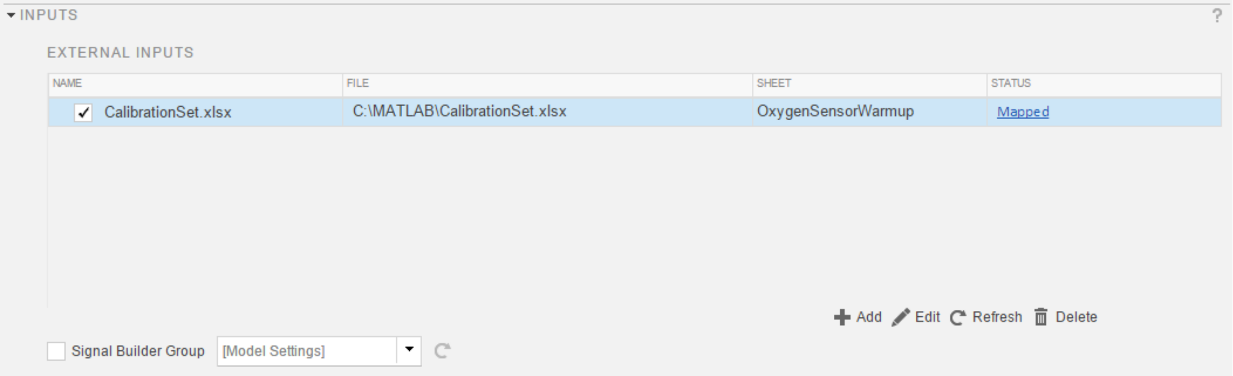
MAPPING STATUS

Successfully mapped inputs.

PORT	BLOCK NAME	MAPPED SIGNAL	STATUS
1	inout/In1		✓

3. Specify a Microsoft Excel file.
4. Select the sheet that contains the input data.
5. If you want to use each sheet to create an input set in the table, select **Create scenarios from each sheet**.
6. Under **Input Mapping**, choose a mapping mode. For more information about mapping modes, see “Map Root Inport Signal Data” (Simulink).
7. Click **Map Inputs**. The **Mapping Status** table shows the port and signal mapping.

   For more information about troubleshooting the mapping status, see “Understand Mapping Results” (Simulink).
8. Click **Apply**.



▾ INPUTS

EXTERNAL INPUTS

NAME	FILE	SHEET	STATUS
☑ CalibrationSet.xlsx	C:\MATLAB\CalibrationSet.xlsx	OxygenSensorWarmup	Mapped

□ Signal Builder Group [Model Settings]

Automate Tests Programmatically

In this section...

List of Functions and Classes

Function	Description
`sltest.testmanager.view`	Open the Simulink Test Manager
`sltest.testmanager.createTestsFromModel`	Generate test cases from a model
`sltest.import.sldvData`	Create test cases from Simulink Design Verifier results
`sltest.testmanager.load`	Load a test file in the Simulink Test Manager
`sltest.testmanager.run`	Run test in the Simulink Test Manager
`sltest.testmanager.copyTests`	Copy test cases or test suites to another location
`sltest.testmanager.moveTests`	Move test cases or test suites to a new location
`sltest.testmanager.report`	Generate report of test results
`sltest.testmanager.clear`	Clear test files from the Simulink Test Manager
`sltest.testmanager.close`	Close the Simulink Test Manager
`sltest.testmanager.clearResults`	Clear results from the Simulink Test Manager
`sltest.testmanager.importResults`	Import Test Manager results file

Function	Description
`sltest.testmanager.exportResults`	Export results set from Test Manager
`sltest.testmanager.getResultSets`	Returns result set objects in Test Manager

Class	Description
sltest.testmanager.TestFile	Create or modify test file
sltest.testmanager.TestSuite	Create or modify test suite
sltest.testmanager.TestCase	Create or modify test case
sltest.testmanager.TestIteration	Create or modify test iteration
sltest.testmanager.ParameterSet	Add or modify parameter set
sltest.testmanager.ParameterOverride	Add or modify parameter override
sltest.testmanager.TestInput	Add or modify test input
sltest.testmanager.CoverageSettings	Modify coverage settings
sltest.testmanager.BaselineCriteria	Add or modify baseline criteria
sltest.testmanager.EquivalenceCriteria	Add or modify equivalence criteria
sltest.testmanager.SignalCriteria	Add or modify signal criteria
sltest.testmanager.Options	View or set test file options
sltest.testmanager.ResultSet	Access results set data
sltest.testmanager.TestFileResult	Access test file results data
sltest.testmanager.TestSuiteResult	Access test suite results data
sltest.testmanager.TestCaseResult	Access test case results data
sltest.testmanager.TestIterationResult	Access test iteration result data
sltest.testmanager.TestResultReport	Customize generated results report

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 Verification and Validation 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);
```

See Also

`sltest.testmanager.report`

Run Multiple Combinations of Tests Using Iterations

In this section...

Test Manager iterations facilitate test cases for multiple sets of data. Use iterations to test many different combinations of parameter sets, external inputs, configuration sets, Signal Builder groups, or baselines. The **Iterations** section of a test case enables you to have many iterations in one centralized location.

There are two ways to set up iterations: tabled and scripted. You can use one or both ways to create iterations in a test case. If you use iterations in a test case where you have specified coverage settings, then the same coverage settings are applied to all iterations in the test case.

Create Table Iterations

The **Table Iterations** section is a quick way to add iterations. The table makes the set of iterations easy to view at a glance. To create iterations:

1. Add parameter sets, external inputs, configuration sets, Signal Builder groups, or baselines to a test case if they are applicable to your tests.
2. To add an iteration to the table manually, click **Add**.
3. By default, the **Parameter Set** and **External Input** columns are visible in the table. To add or remove columns, click the ✚ button, and select a column from the list.
4. In the iteration row, select the column cell you want to use to change the test setting. For example, if you want to have an iteration with a parameter set, click the cell below **Parameter Set**, and select the parameter set from the drop-down list.

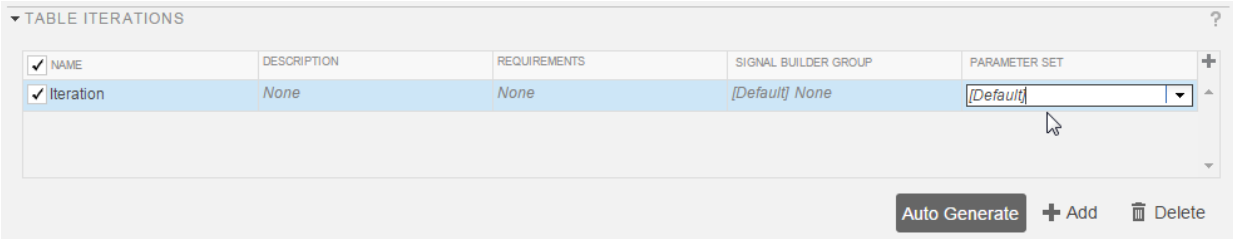
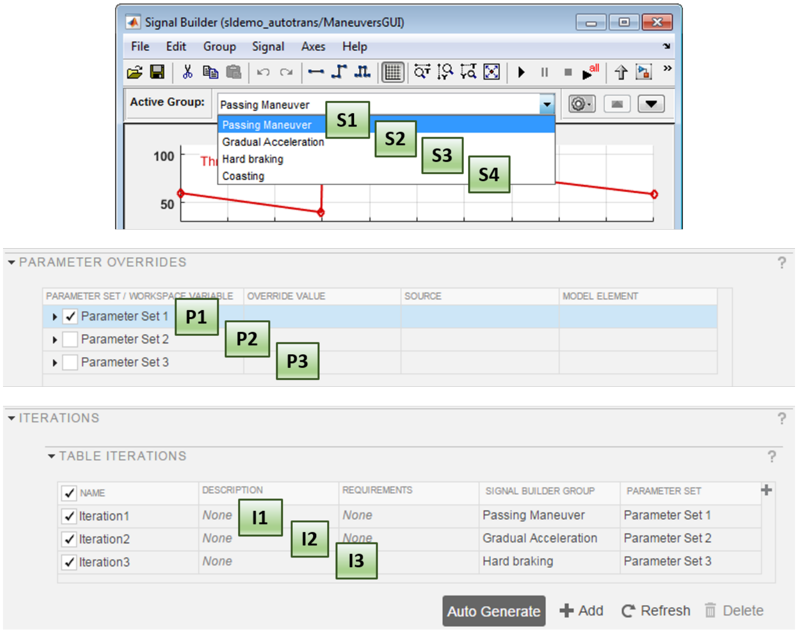


TABLE ITERATIONS

☑ NAME	DESCRIPTION	REQUIREMENTS	SIGNAL BUILDER GROUP	PARAMETER SET
☑ Iteration	None	None	[Default] None	[Default]

Autogenerated iteration combinations are ordered in lockstep. Lockstep means that each iteration is formed using sequential pairings of test case settings. For example, the model `sldemo_autotrans` has a Signal Builder block with four signal groups, labeled in the figure as S1, S2, S3, and S4. If you use this model in a test case with three parameter sets, labeled as P1, P2, and P3, then the Test Manager generates three iterations. Generated iterations are limited to the minimum number settings between Signal Builder groups and parameter sets, which is three. Each iteration, labeled as I1, I2, and I3, contains one Signal Builder group with the corresponding parameter set. The Signal Builder group and parameter set are matched in the order that they are listed in the Signal Builder block or parameter set section, respectively.



Signal Builder (sldemo_autotrans/ManeuversGUI)

Active Group: Passing Maneuver

- Passing Maneuver
- Gradual Acceleration
- Hard braking
- Coasting

PARAMETER OVERRIDES

PARAMETER SET / WORKSPACE VARIABLE	OVERRIDE VALUE	SOURCE	MODEL ELEMENT
☑ Parameter Set 1			
□ Parameter Set 2			
□ Parameter Set 3			

ITERATIONS

TABLE ITERATIONS

☑ 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		Hard braking	Parameter Set 3

In the table iterations, `Default [None]` means that the iteration does not change the test case setting. The test iteration setting is the same as what is specified in the test case.

View Table Iterations

To see a list of iterations from the **Table Iterations** section, click **Show Iterations**. The list includes table iterations and scripted iterations.

Generate Table Iterations

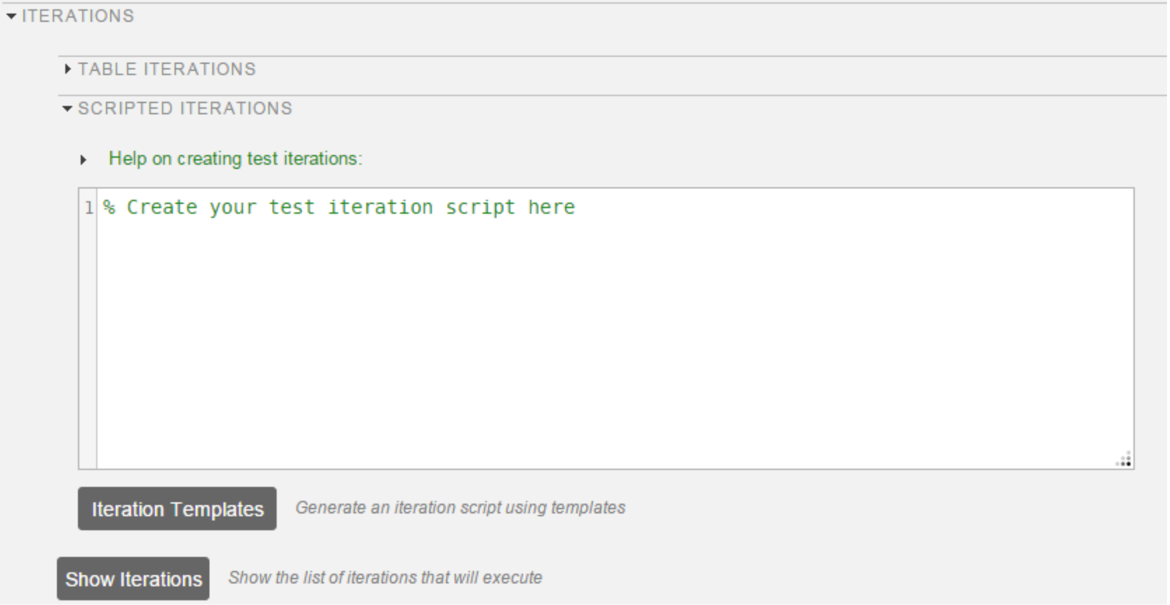
If you have test case settings that you want to transform into test iterations, then you can use the **Auto Generate** button. You can choose to generate iterations for different test case sections. If you select multiple sections in the dialog box, then the Test Manager combines iterations, and lockstep ordering applies.

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.
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 Option	Purpose
	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

In the scripted iterations section of the test case, you can customize your own set of iterations using a programmatic workflow. You can define your own parameter sets, customize the order of the iterations, create your own Monte Carlo script, and more. Scripted iterations are generated at run time when a test executes. Enter the script into the Scripted Iterations section text box.


ITERATIONS
TABLE ITERATIONS
SCRIPTED ITERATIONS
Help on creating test iterations:

```
% Create your test iteration script here
```

Iteration Templates
Generate an iteration script using templates
Show Iterations
Show the list of iterations that will execute

Iteration Script Components

An iteration script must have certain components to execute the tests. The basic iteration script contains three elements: an iteration object, an iteration setting, and the iteration registration. This script iterates over a single signal builder groups. This example is not practical, but it is meant to illustrate the anatomy of an iteration script.

```
%% 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. In practice, you iterate over numerous settings, such as multiple Signal Builder groups. If you take the stripped-down iteration script and put it into a loop, you can iterate over all Signal Builder groups in the test case.

```
%% 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 6-33.

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.

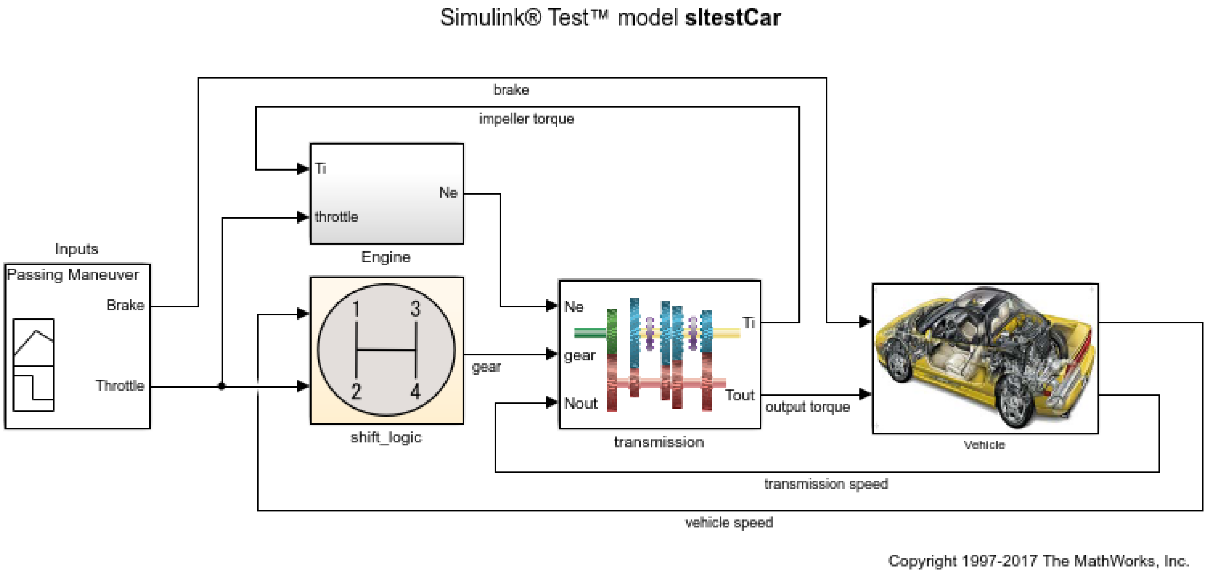
3 To generate a table that gives a preview of the iterations that execute when you run the test case, click **Show Iterations**.

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 **Basleine 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.

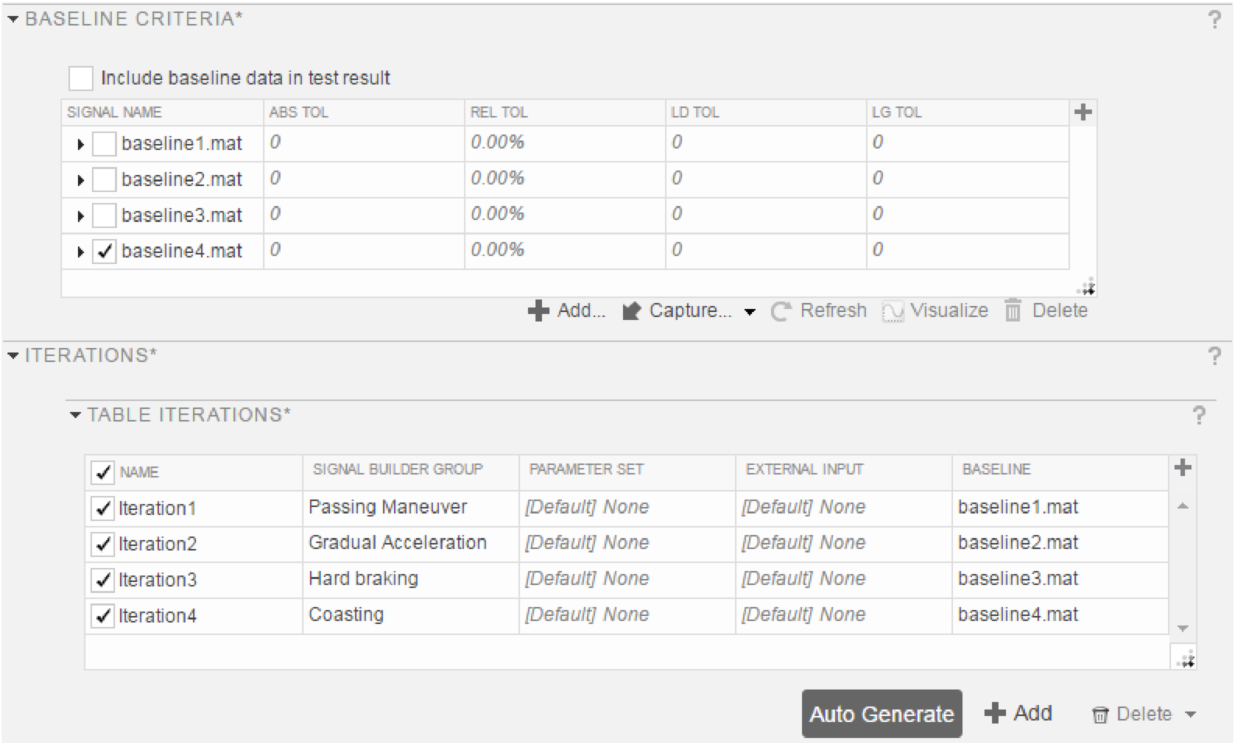
6. Add iterations for the test case:

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.


BASELINE CRITERIA*

□ 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

ITERATIONS*

TABLE ITERATIONS*

☑ 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. In this example of a parameter sweep, the number of Signal Builder groups and parameter values is the same. Each iteration has one Signal Builder group and one parameter value for a total of four iterations.

```
%% Iterate over all Signal Builder Groups and Parameters

% Determine the number of possible iterations
numSteps = length(sltest_signalBuilderGroups);

% Set up the parameter values to sweep over
IeiValues = [0.021,0.022,0.022,0.023];

% Create each iteration
```

```
for k = 1 : numSteps
    % Set up a new iteration object
    testItr = sltestiteration;

    % Set Signal Builder iteration setting
    setTestParam(testItr,'SignalBuilderGroup',sltest_signalBuilderGroups{k});

    % Set value of lei (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
```

See Also

sltest.testmanager.TestIteration

Related Examples

- “Automate Tests Programmatically” on page 6-27

Collect Coverage in Tests

In this section...

If you use Simulink Verification and Validation to generate model and code coverage, then you can also apply coverage collection to your test cases. If you turn on coverage collection in a test file, test suite, or test case, then the test case runs in the Test Manager, collects coverage, and generates an aggregated report of the coverage in the test results.

For example, to test a model for coverage, turn on coverage at the test-file, test-suite, or the individual test-case level. If you set coverage settings at the test-file level, then the test suite and test cases in the test file inherit the coverage settings. At the test case and test-suite level, you can turn coverage collection on or off, but you cannot adjust the coverage metrics if they are specified at the test-file level.

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 2-41.

Coverage results association depends on test harness – main model synchronization:

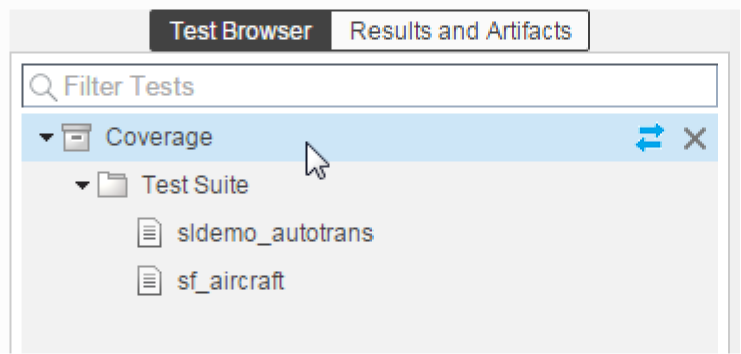
- If the test harness is configured to synchronize the component under test when you open or close the harness, coverage results from the test harness are associated with the main model. When you close the test harness, the coverage results remain active in memory. You can aggregate coverage with additional 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`.

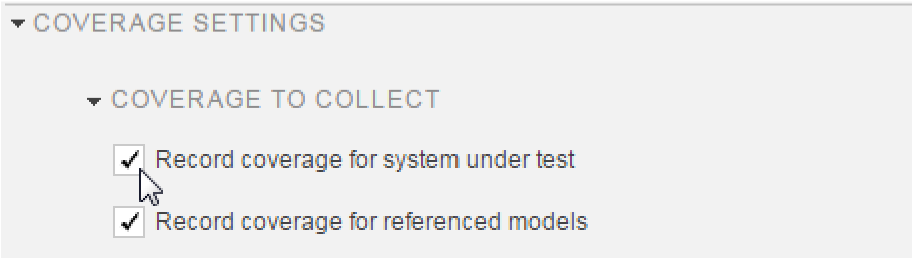
Enable and Collect Coverage for a Test File

This procedure outlines how to enable coverage collection when you run a test file, view coverage results in the Test Manager, and trace coverage results from the Test Manager to the model.

1 Select the test file in the **Test Browser** pane.



2 Expand the **Coverage Settings** section in the test file and select **Record coverage for system under test**. To collect coverage for referenced models, select **Record coverage for referenced models**.



☑ Record coverage for system under test

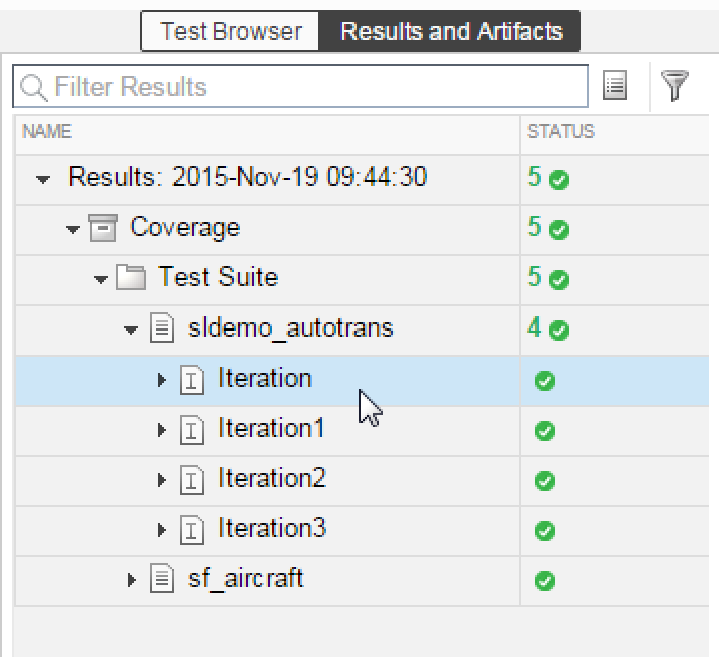
☑ Record coverage for referenced models

3 Select coverage metrics. For more information about the types of model coverage, see "Types of Model Coverage" (Simulink Verification and Validation).

4 Run the test file.

5 To view the collected coverage results, select a test case result in the **Results and Artifacts** pane and expand the **Coverage Results** section of the test case result.

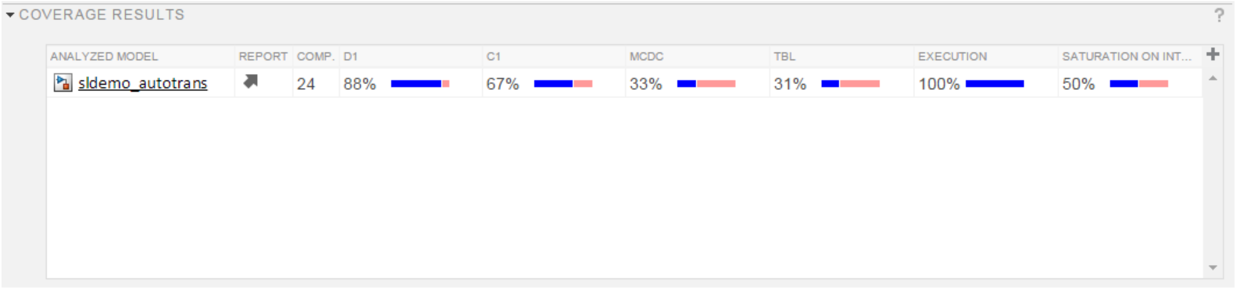
This example shows results for an iteration of a test case.



Test Browser | Results and Artifacts

Filter Results

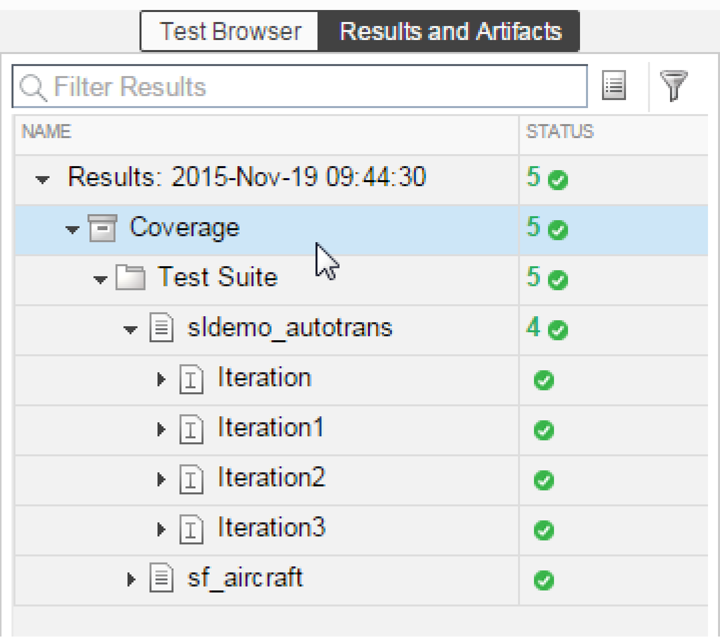
NAME	STATUS
Results: 2015-Nov-19 09:44:30	5 ✓
Coverage	5 ✓
Test Suite	5 ✓
sldemo_autotrans	4 ✓
Iteration	✓
Iteration1	✓
Iteration2	✓
Iteration3	✓
sf_aircraft	✓



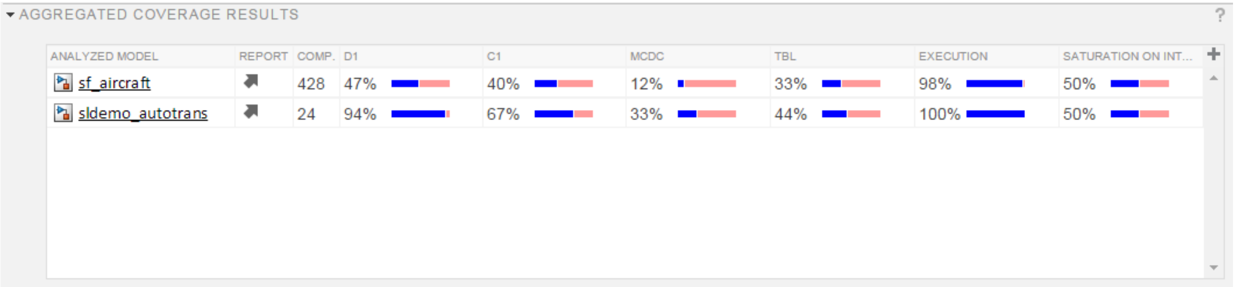
COVERAGE RESULTS

ANALYZED MODEL	REPORT	COMP.	D1	C1	MCDC	TBL	EXECUTION	SATURATION ON INT...
sldemo_autotrans	↗	24	88%	67%	33%	31%	100%	50%

6 To view the aggregated coverage for a test case, test suite, or test file, select the corresponding result, then expand the **Aggregated Coverage Results** section of the test file result. At the test suite and test file result level, the coverage is grouped and aggregated by model.

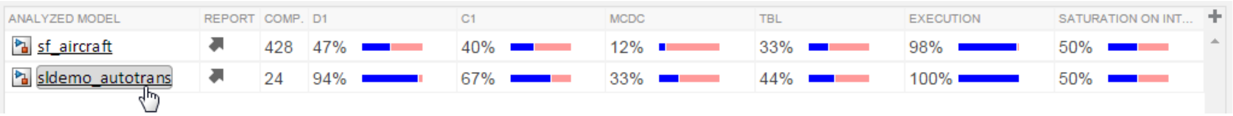


NAME	STATUS
Results: 2015-Nov-19 09:44:30	5 ✓
Coverage	5 ✓
Test Suite	5 ✓
sldemo_autotrans	4 ✓
Iteration	✓
Iteration1	✓
Iteration2	✓
Iteration3	✓
sf_aircraft	✓



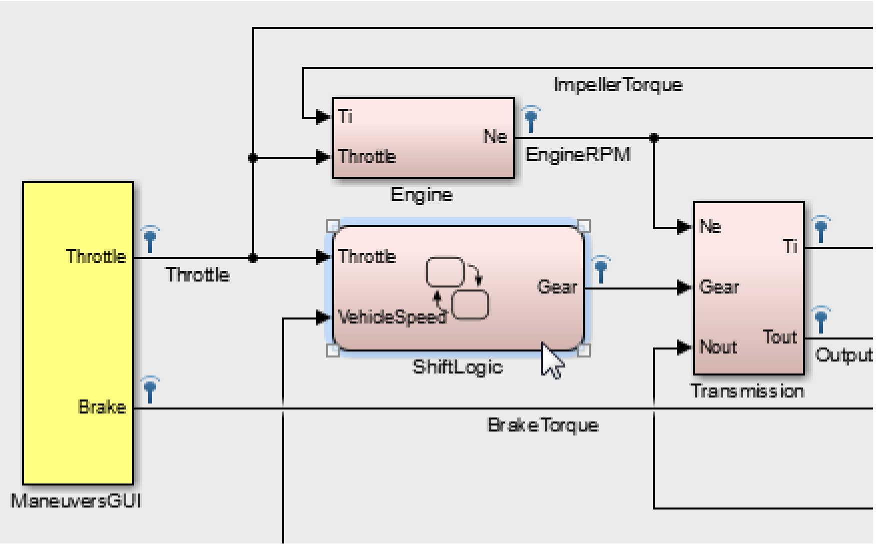
ANALYZED MODEL	REPORT	COMP.	D1	C1	MCDC	TBL	EXECUTION	SATURATION ON INT...
sf_aircraft		428	47%	40%	12%	33%	98%	50%
sldemo_autotrans		24	94%	67%	33%	44%	100%	50%

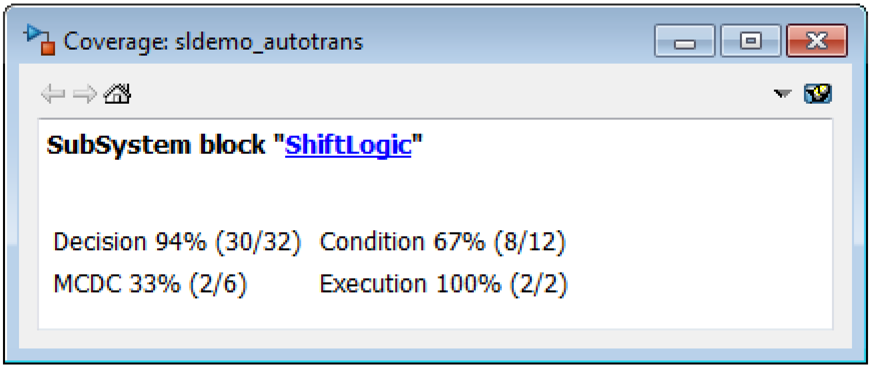
7 To view the coverage results graphically in the model, click the model name link in the coverage results table.



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%

Once the model opens, you can select parts of the model to see the coverage data.





Coverage: sldemo_autotrans

SubSystem block "ShiftLogic"

Decision 94% (30/32) Condition 67% (8/12)

MCDC 33% (2/6) Execution 100% (2/2)

To view the coverage results programmatically, see “Automate Tests Programmatically” on page 6-27 and the sltest.testmanager.CoverageSettings class.

See Also

“Specify Coverage Options” (Simulink Verification and Validation) | “Perform Functional Testing and Analyze Test Coverage” on page 9-9

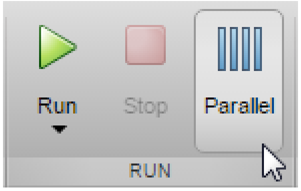
Run Tests Using Parallel Execution

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.

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).
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.

When Will 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.
- A large number of long-running tests, such as iterations.

See Also

`sltest.testmanager.run`

Related Examples

- "Clusters and Clouds" (Parallel Computing Toolbox)

Apply Tolerances to Test Criteria

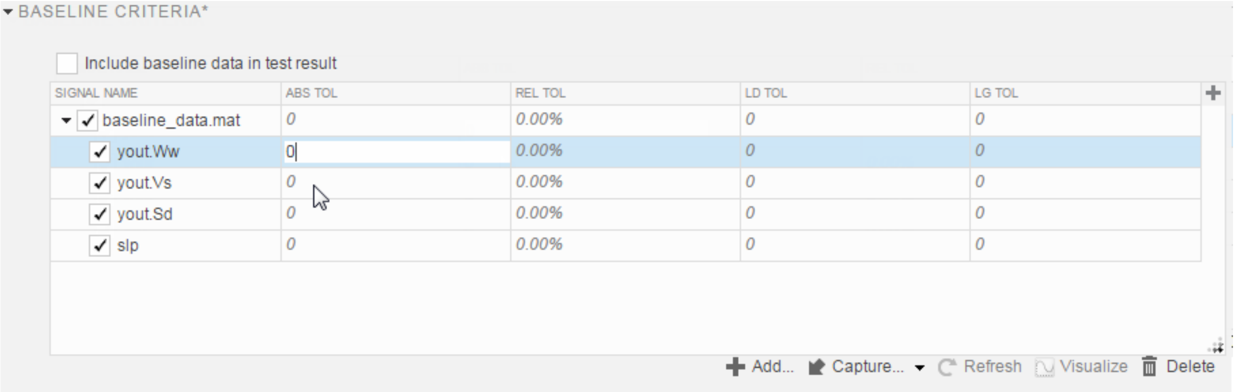
In this section...
"Modify Criteria Tolerances" on page 6-51
"Change Leading Tolerance in a Baseline Comparison Test" on page 6-51

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.



□ Include baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL	LD TOL	LG TOL
✓ baseline_data.mat	0	0.00%	0	0
✓ yout.Ww	0	0.00%	0	0
✓ yout.Vs	0	0.00%	0	0
✓ yout.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

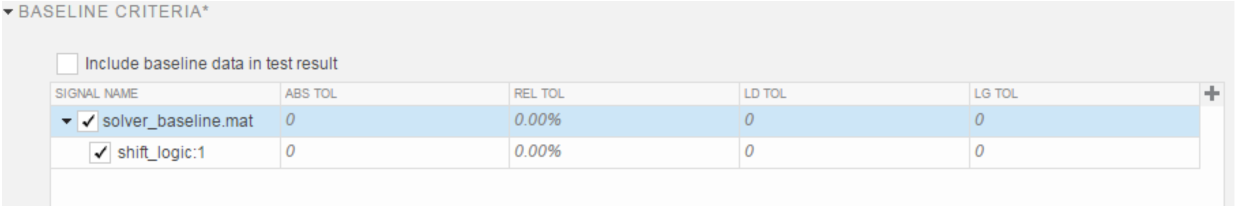
Specify a tolerance when the difference between results falls in a range you consider acceptable. Suppose that your model under test uses a particular solver. Solvers are sometimes updated from one release to the next, and new solvers also become available. If you use an updated solver or change solvers, 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**. Name the baseline `solver_baseline` and click **Save**.

   After you save the baseline MAT-file, the model runs and the baseline criteria appear in the table. Each default tolerance is 0.



BASELINE CRITERIA*

□ Include baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL	LD TOL	LG TOL
☑ solver_baseline.mat	0	0.00%	0	0
☑ shift_logic:1	0	0.00%	0	0

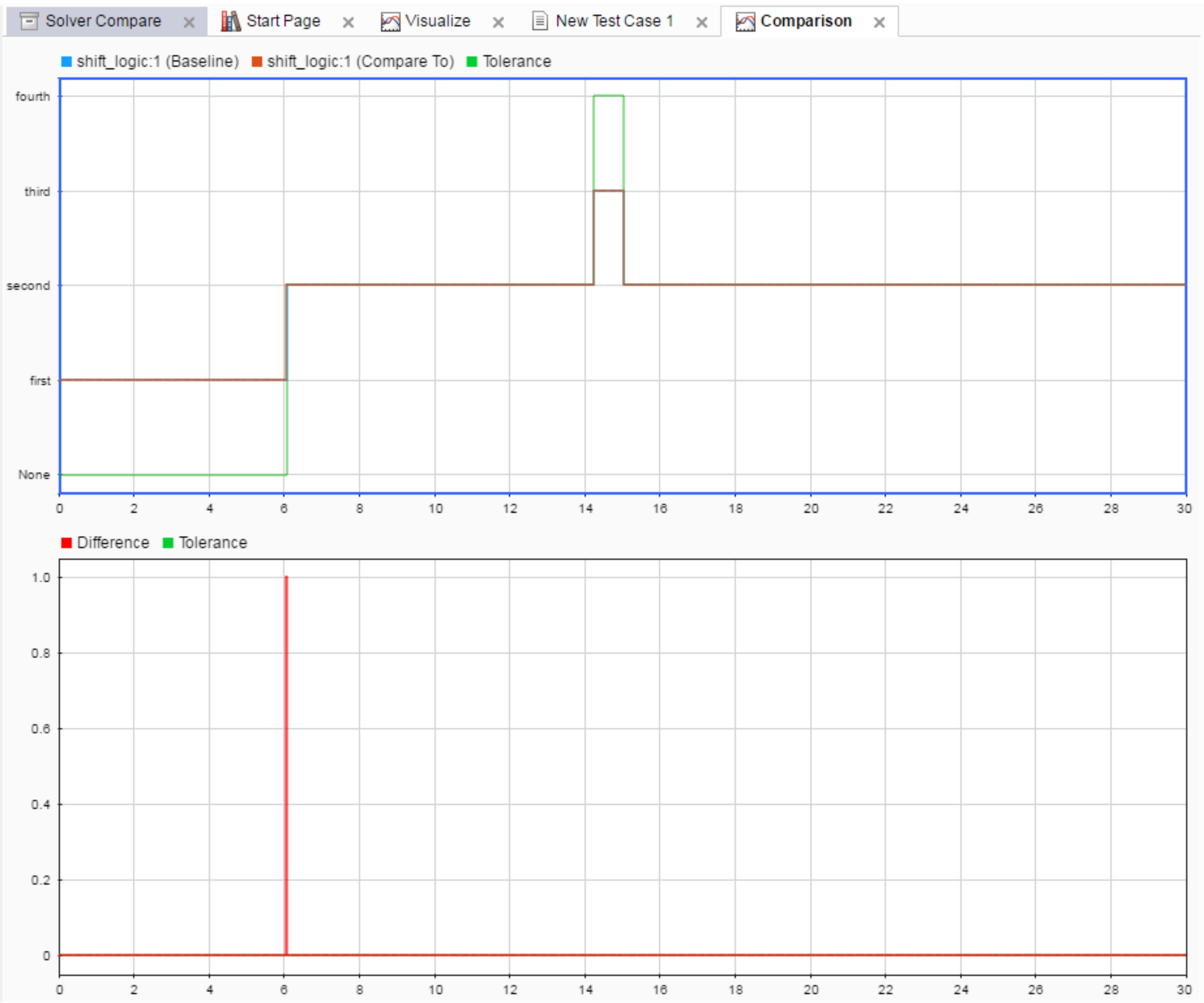
Change Solvers and Run the Test Case

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.

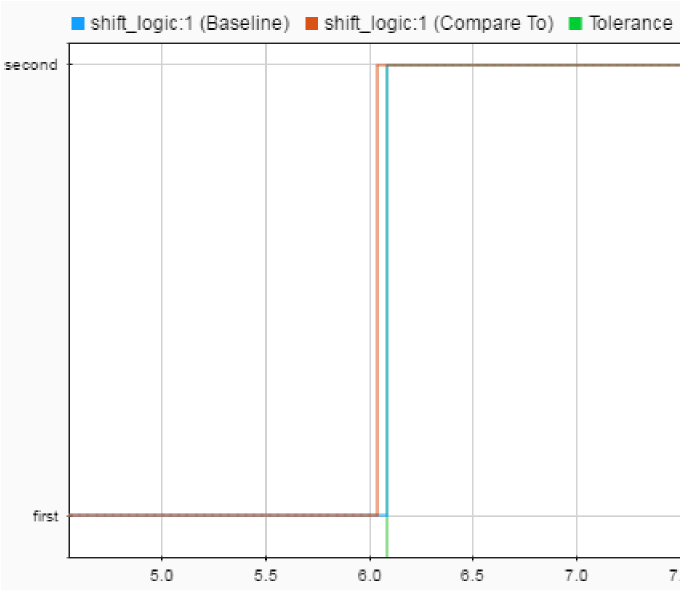
1 In the model, change the solver to `ode-1`.
2 In the Test Manager, with the `Solver Compare` test file selected, click **Run**.

   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.

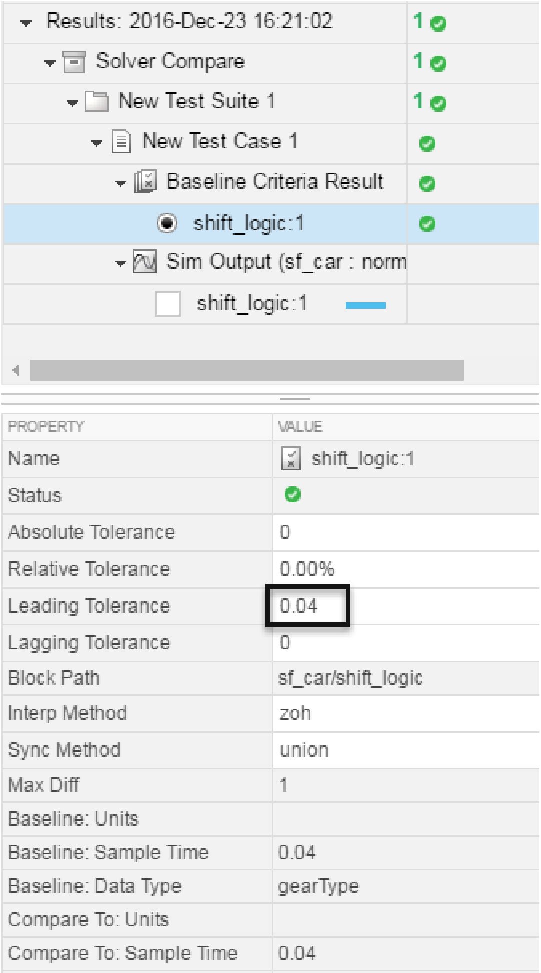


Preview and Set a Leading Tolerance Value

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`.



Results: 2016-Dec-23 16:21:02	1 ✓
Solver Compare	1 ✓
New Test Suite 1	1 ✓
New Test Case 1	✓
Baseline Criteria Result	✓
● shift_logic:1	✓
Sim Output (sf_car : norm	
□ shift_logic:1	

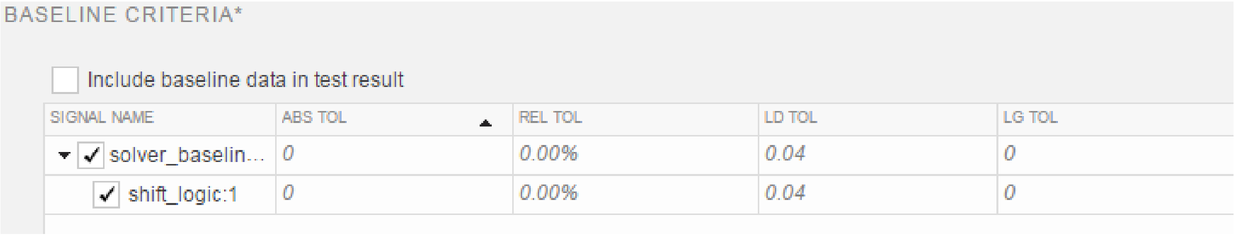
PROPERTY	VALUE
Name	shift_logic:1
Status	✓
Absolute Tolerance	0
Relative Tolerance	0.00%
Leading Tolerance	0.04
Lagging Tolerance	0
Block Path	sf_car/shift_logic
Interp Method	zoh
Sync Method	union
Max Diff	1
Baseline: Units	
Baseline: Sample Time	0.04
Baseline: Data Type	gearType
Compare To: Units	
Compare To: Sample Time	0.04

When you change this value, the status of the results changes to show that the failed tests will 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 **LD TOL** (leading tolerance) 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.



BASELINE CRITERIA*

□ Include baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL	LD TOL	LG TOL
✓ solver_baselin...	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

- "Test Model Output Against a Baseline" on page 6-9

Test Manager Limitations

In this section...

Simulation Mode

There are some limitations for the simulation mode in test cases:

- The **System Under Test** cannot be in fast restart or external mode for test execution.
- A test that is running with the **System Under Test** simulation mode set to **Rapid Accelerator** cannot be stopped using **Stop** on the Test Manager toolstrip. To stop the test, enter **Ctrl+C** in the MATLAB command prompt.
- If you run a test using parallel execution in rapid accelerator mode, streamed signals do not show up in the Test Manager.

Callback Scripts

The test case callback scripts are not stored with the model and do not override Simulink model callbacks. Test case callback scripts have some limitations:

- The Test Manager cannot stop the execution of an infinite loop inside a callback script. 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.

Protected Models

You cannot specify a protected model as the model used for a test case in the **System Under Test** section.

Parameter Overrides

The Test Manager displays only top-level system parameters from the system under test.

Breakpoints

Breakpoints in Simulink and Stateflow are not supported and interrupt test execution without warning.

Highlight in Model

If you use parallel test execution to run your tests, then you cannot use the **Highlight in Model** button for `verify` signals.

Test Sections

In this section...

Select a test file, suite or case in the **Test Browser** pane to access the test sections. For information on which test case to use for your application, see "Introduction to the Test Manager" on page 5-2.

Set Preferences to Hide Unused Test Sections

To simplify the Test Manager layout, you can hide unused sections in your test case, test suite, or test file. This is useful if you need to update or review several test cases in the Test Manager, or if you normally do not use certain sections. If you enter information in a section, the section displays regardless of how you set the preferences.

1 In the toolbar, 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 populated sections, clear all selections in the **Preferences** dialog box.
4. Click **OK**.

Also see `sltest.testmanager.getpref` and `sltest.testmanager.setpref`.

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 6-97.

Description

To add descriptive text to your test case, test suite, or test file, expand the section and double-click the text box below **Description**.

Requirements

You can create, edit, and delete requirements traceability links for a test case, test suite, or test file in the **Requirements** section if you have a Simulink Verification and Validation license. To add requirements links:

1. Click the **Add** button ✚ Add .
2. In the Link Editor dialog box, click **New** to add a requirement link to the list.
3. Type the name of the requirement link in the **Description** box.
4. Click **Browse** and locate the requirement file. Click **Open**. For more information on supported requirements document types, see “Supported Requirements Document Types” (Simulink Verification and Validation).
5. Click **OK**. The requirement link appears in the Requirements list if a document is specified in the Link Editor.

If you have a section of a document open and ready to add as a requirement, then you can add it quickly. Highlight the section you want to add as a requirement, click the **Add** button arrow, and select the section type.

For more information about the Link Editor, see “Requirements Traceability Link Editor” (Simulink Verification and Validation).

System Under Test

Specify the model you want to test in the **System Under Test** section. To use the current model that is in focus, click the **Use current model** button.

Note: The model must be available on the path to run the test case. You can set the path programmatically using the preload callback. See “Callbacks” on page 6-62.

Specifying a new model in the **System Under Test** section can cause the model information to be obsolete. 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 be used for the test case. If a test harness has been added or removed from a 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 2-15.

Simulation Settings

You can override the **System Under Test** simulation settings such as the simulation mode, start time, stop time, and initial state.

Parameter Overrides

You can specify parameter values in the test case to override the parameter values in the model workspace, data dictionary, or base workspace in the **Parameter Overrides** section. Parameters are grouped into sets. Parameter sets and individual parameters overrides can be turned on or off by selecting or clearing 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, press the **Refresh** button ↻ in the dialog box to update the list.
2. Select the parameter you want to override.
3. Click **OK** to add the parameter to the parameter set.
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 about parameter overrides, see .

Callbacks

Test-File Level Callbacks

There are two callback scripts 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

There are two callback scripts 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

There are three callback scripts available in each test case that execute at different times during a test:

- Pre-load: runs before the model loads and any model callbacks.
- Post-load: runs after the model loads and the `PostLoadFcn` model callback.
- Cleanup: runs after simulations and all model callbacks.

Click the **Run** button ▷ next to **Pre-Load**, **Post-Load**, or **Cleanup** to run only that callback script.

See “Test Manager Limitations” on page 6-57 for the limitations of callback scripts inside test cases. For information on Simulink model callbacks, see “Model Callbacks” (Simulink).

There are predefined variables available to you 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.

Inputs

You can override inputs to the system under test. You can use inputs from signal builder groups in the model, or you can use external inputs from MAT-files or Microsoft Excel files.

- To use inputs from a Signal Builder block group in the model or test harness, select **Signal Builder Group**, the select the group from the drop-down list.
- You can use only one external input set in the **External Inputs** table to run when the test case executes. See “Identify Signal Data to Import and Map” (Simulink) for more information on supported file formats.
- For an example of how to use external inputs, see “Use External Inputs in Test Cases” on page 6-24. For more information on the Root Inport Mapper tool, see “Map Root Inport Signal Data” (Simulink)

.

Simulation Outputs

Use the **Simulation Outputs** section to add signal outputs to your test results. Signals already logged in your model or test harness also show up in the results. To log additional signals,

1. Click the **Add** button below the **Logged Signals** table. The system under test opens.

   To add a signal set, click the **Add** drop-down arrow and select **Signal Set**.
2. In the Simulink canvas, highlight signals by clicking individual signal lines, or area-selecting multiple signals.
3. In the **Signal Selection** dialog box, select signals of interest and click **Add**.

Simulation output properties save with the test case; selecting additional signals does not change your model or test harness.

Configuration Settings

You can override the **System Under Test** configuration settings.

Note: Selecting **Override model settings** overrides the output settings in the model or test harness configuration settings.

Simulation 1 and Simulation 2

The Simulation 1 and Simulation 2 sections in the equivalence test case are the same templates. The system under test from Simulation 1 and Simulation 2 are compared to each other using the signal data defined under **Equivalence Criteria**.

Equivalence Criteria

This test case section appears only in an equivalence test case. The equivalence criteria is a set of signal data that is compared between Simulation 1 and Simulation 2 in an equivalence test case. You can specify both absolute and relative tolerances for individual signals or the entire criteria set. Tolerances can be specified in this section to regulate pass-fail criteria of the test.

Click **Capture** to run the system under test in Simulation 1 and identify signals for equivalence criteria. Signals in the model marked for streaming and logging are captured.

Note: If you stream the same signal in the system under test of Simulation 1 and Simulation 2, and do not capture any equivalence criteria, then the streamed signals are compared in the equivalence criteria result. However, if you do capture equivalence criteria and no signals are selected, then nothing is compared when the test case executes.

For an example about how to use an equivalence test case and criteria, see .

Baseline Criteria

This test case section appears only in a baseline test case. You can use signal data from a MAT-file or Microsoft Excel file. Microsoft Excel files must use formatting specified by the Root Inport Mapper tool. For more information, see “Map Root Inport Signal Data” (Simulink). Only the first sheet of the Microsoft Excel file is read for baseline criteria.

To capture streamed and logged signal data from the **System Under Test**, click **Capture** to compile and run the system. You are asked to save the signal data to a MAT-file.

Tolerances can be specified in this section to determine the pass-fail criteria of the test case. You can specify both absolute and relative tolerances for individual signals or the entire baseline criteria set. When the baseline test case executes, signals in the model marked for streaming and logging are captured and compared to the baseline criteria. To see tolerances used in an example for baseline criteria, see “Test Model Output Against a Baseline” on page 6-9.

Custom Criteria

This section provides 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 “Apply Custom Criteria to Test Cases” on page 6-73. 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 6-85.

Iterations

This test case section is used to generate test iterations for multiple combinations of test settings. Iterations are helpful for Monte Carlo or parameter sweep tests. For more information about test iterations, see “Run Multiple Combinations of Tests Using Iterations” on page 6-33.

Coverage Settings

This test section lets you 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 6-43.

Test File Options

Close all 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 6-85.

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 7-9 and “Customize Test Reports” on page 7-14.

Test Models Using Inputs Generated by Simulink Design Verifier

In this section...
"Overall Workflow" on page 6-68
"Test Case Generation Example" on page 6-69

Using Simulink Design Verifier, you can generate tests that replicate design errors, achieve test objectives, or exercise your model to meet coverage criteria. Over the course of developing your model and generating code, you repeatedly exercise your model and code with these test inputs. You can simplify repeated testing using Simulink Test to automatically create test cases that use inputs generated using Simulink Design Verifier analysis.

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 in the Test Manager. 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 easily 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.

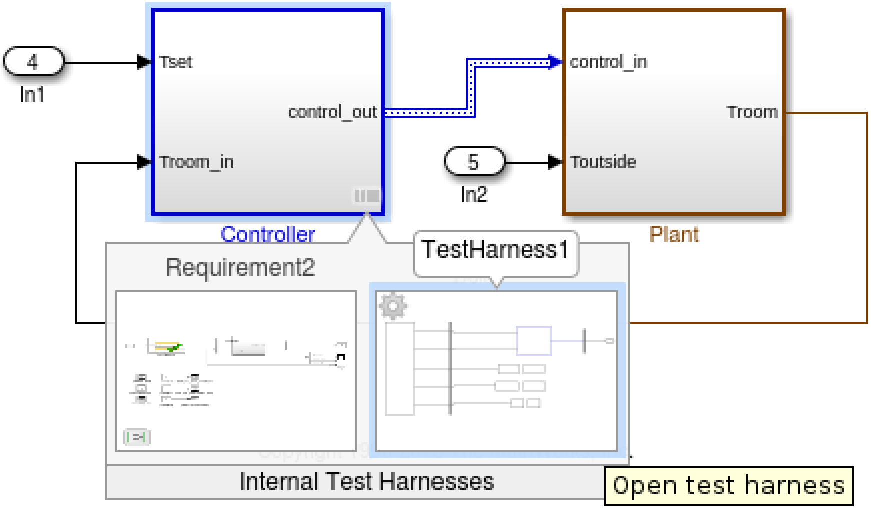
1. Open the model.

   ```
   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**.

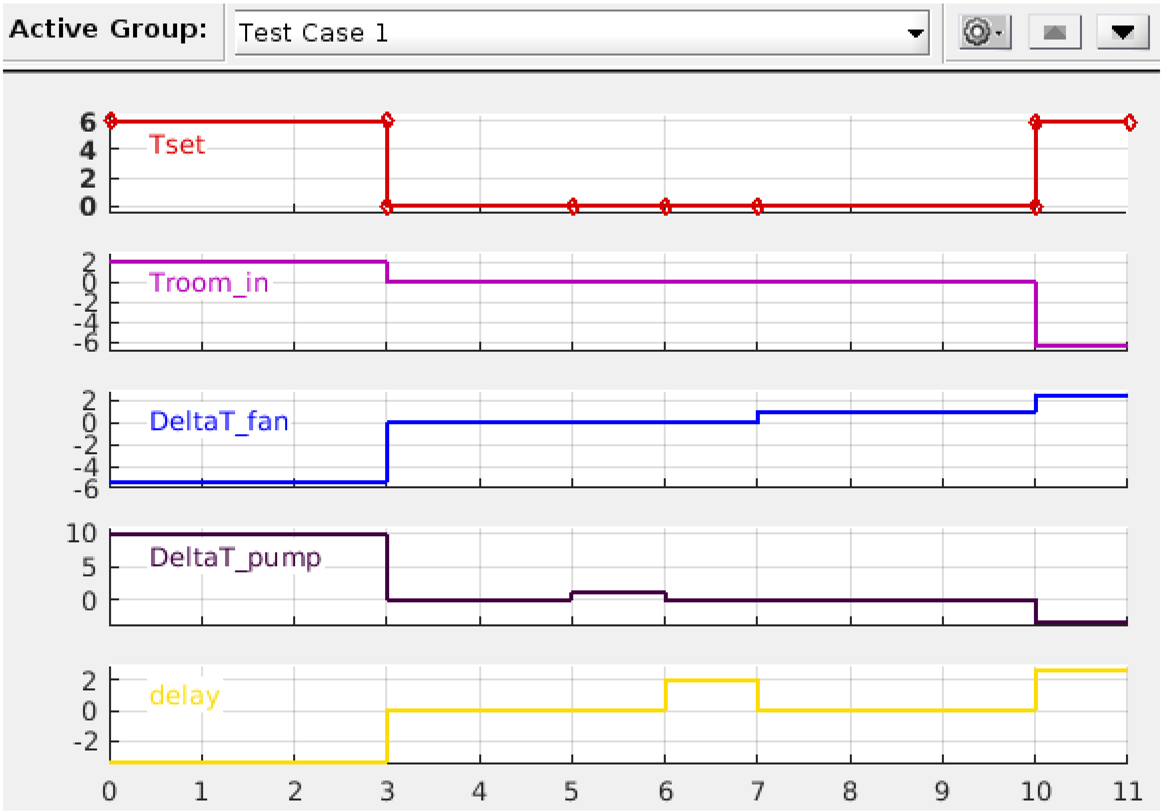
   Simulink Design Verifier generates tests for the component.
4. In the results summary window, click **Export test cases to Simulink Test**.
5. In the Export Design Verifier Test Cases dialog box, enter:

   - 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**.

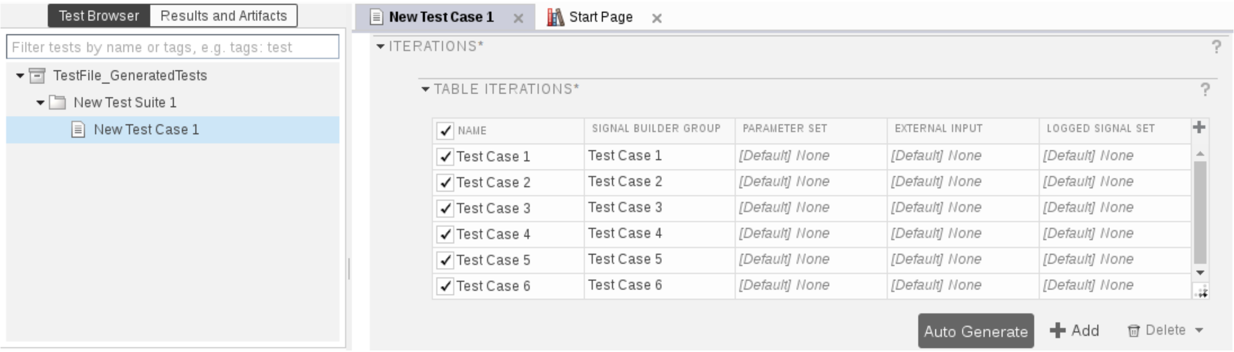
A new test file is created in the working folder, and a test harness is added to the main model, owned by the `Controller` subsystem. Click the harness badge to preview the new test harness.



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`

Apply Custom Criteria to Test Cases

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 Simulation Using Logical Statements" on page 3-25.

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 – Failures appear in the test results and other assessments are evaluated. Use verifications for general assessments, such as checking simulation against expected outputs.

  Example: `test.verifyEqual(lastValue,0)`
- Assertions – Use assertions for conditions that render the criteria invalid. Failures appear in the test results and the custom criteria script evaluation exits.

  Example: `test.assertEqual(lastValue,0)`.

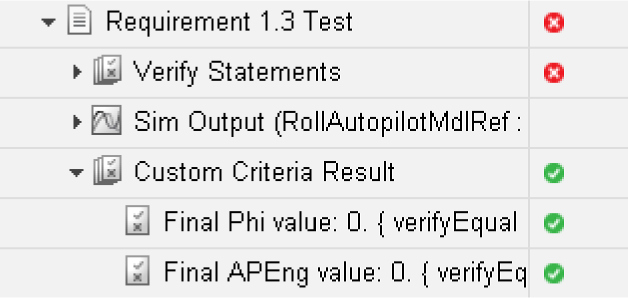
Define a Custom Criteria Script

This example shows how to create a custom criteria script for an autopilot test case.

1. Open the test file.

   ```
   open AutopilotTestFile.mldatx
   ```
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 Smiulink 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

```
open AutopilotTestFile.mldatx
```

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

Use a custom criteria script to verify the damping ratio in a test case that simulates the flutter suppression system of a wing.

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 arodynamic forces on the wing.

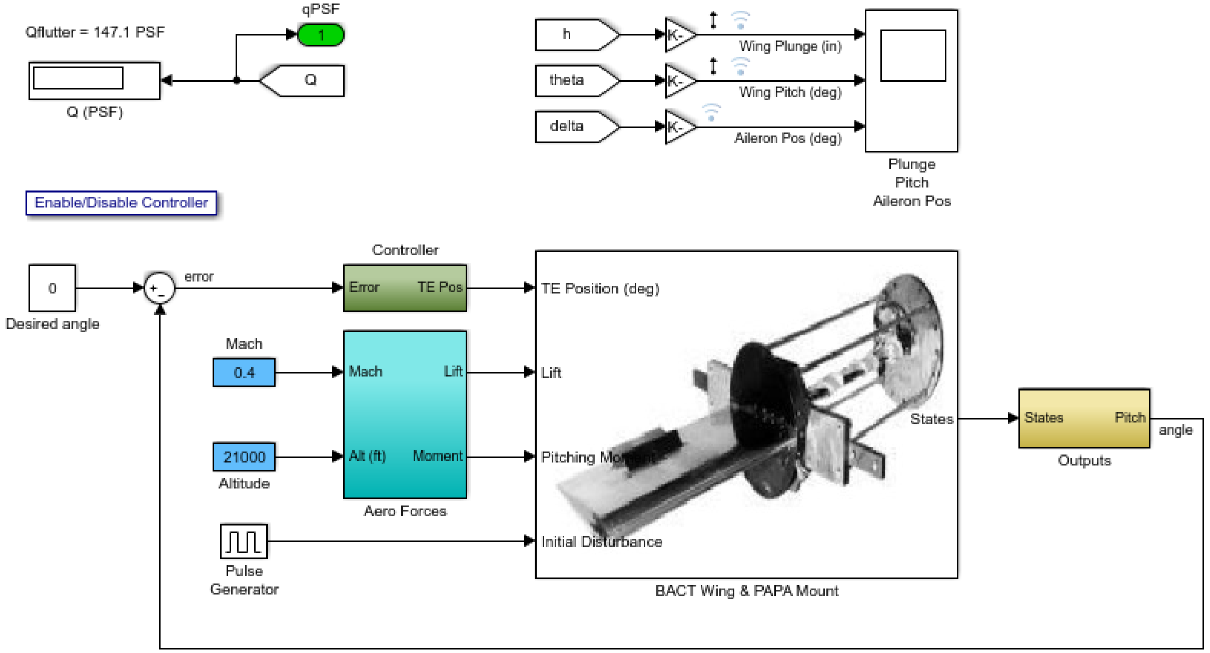
The test iterates over 16 combinations of `Mach` and `Altitude`. The test case uses custom criteria with Curve Fitting Toolbox™ to find the peaks of the wing pitch, and determine the damping ratio. If the damping ratio is not greater than zero, the assessment fails.

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
```

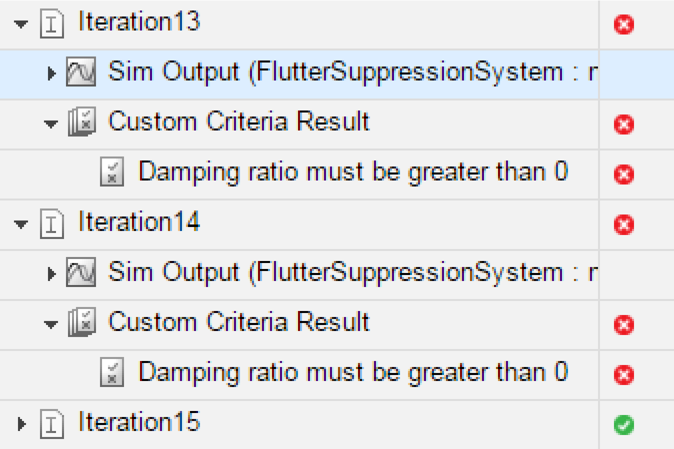
```
fittedModel = exponentialFitAndPlot(L, rn);
delta = fittedModel.d;

% 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.

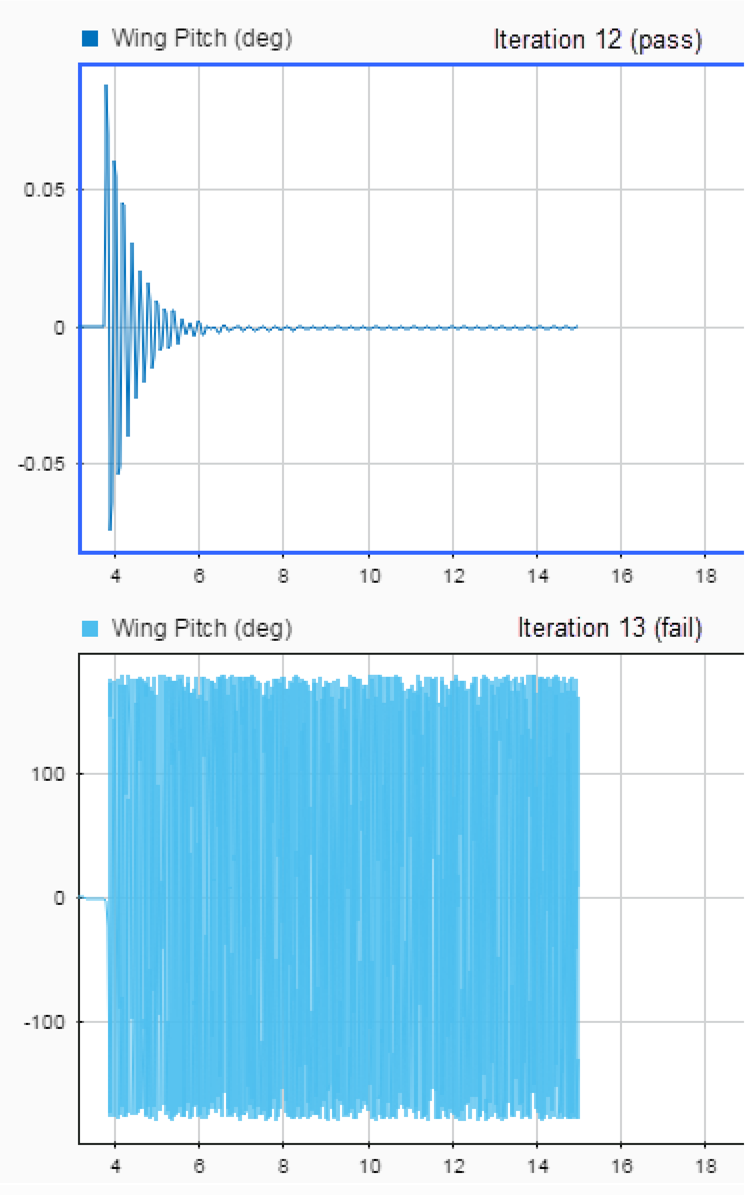
```
results = sltest.testmanager.run
```

```
results = 

  ResultSet with properties:

                   Name: 'Results: 2017-Feb-24 12:32:15'
              NumPassed: 14
              NumFailed: 2
            NumDisabled: 0
          NumIncomplete: 0
               NumTotal: 16
     NumTestCaseResults: 0
    NumTestSuiteResults: 0
     NumTestFileResults: 1
                Outcome: Failed
              StartTime: '2017-Feb-24 12:32:15'
               StopTime: '2017-Feb-24 12:34:12'
               Duration: 117
        CoverageResults: []
```



```
▾ 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: ''
```

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: [1×1 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
```

Related Examples

- “Test Models Using MATLAB Unit Test” on page 6-88

- “Create, Store, and Open MATLAB Figures” on page 6-85

Create, Store, and Open MATLAB Figures

In this section...
"Create a Custom Figure for a Test Case" on page 6-85
"Include Figures in a Report" on page 6-87

You can create figures using MATLAB commands to include with test results and reports. You can enter the commands in any section in your test case that accepts MATLAB code. These include the test case **Custom Criteria** section and any of the 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

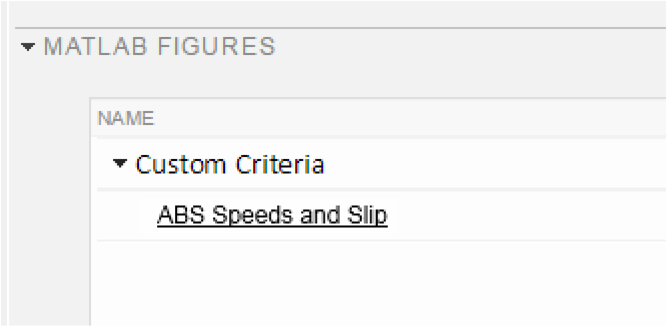
In this example, add code that creates a figure to the **Custom Criteria** section of a test case. To access the figure from the test results, set options on the test file.

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 preformed, 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 7-9

Test Models Using MATLAB Unit Test

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™.

To use MATLAB Unit Test, create a TestSuite from the Simulink Test file. To customize the test execution, such as for CI, create a TestRunner. Running the test produces a TestResult object. For CI, running the test can also stream results to a file.

In this section...

Considerations

When running tests using MATLAB Unit Test, consider the following:

- Test hierarchy from Simulink Test is not converted by MATLAB Unit Test. All tests in a `TestSuite` are contained in a flat hierarchy.
- If you disable a test in the Test Manager, the test is filtered using MATLAB Unit Test, and the result reflects a failed assumption.
- Fast restart is not supported for the MATLAB Unit Test framework.

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);
```

```
Running AutopilotTestFile > Basic Design Test Cases

================================================================================
Verification failed in AutopilotTestFile > Basic Design Test Cases/Requirement 1.3 Tes

    ---------------------
    Framework Diagnostic:
    ---------------------
    Failed criteria: Verification
================================================================================
.
Done AutopilotTestFile > Basic Design Test Cases
__________

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
     ================================================================================'
```

Enable warnings.

```
warning on Stateflow:Runtime:TestVerificationFailed;
warning on Stateflow:cdr:VerifyDangerousComparison;
```

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

Test a Model for Continuous Integration Systems

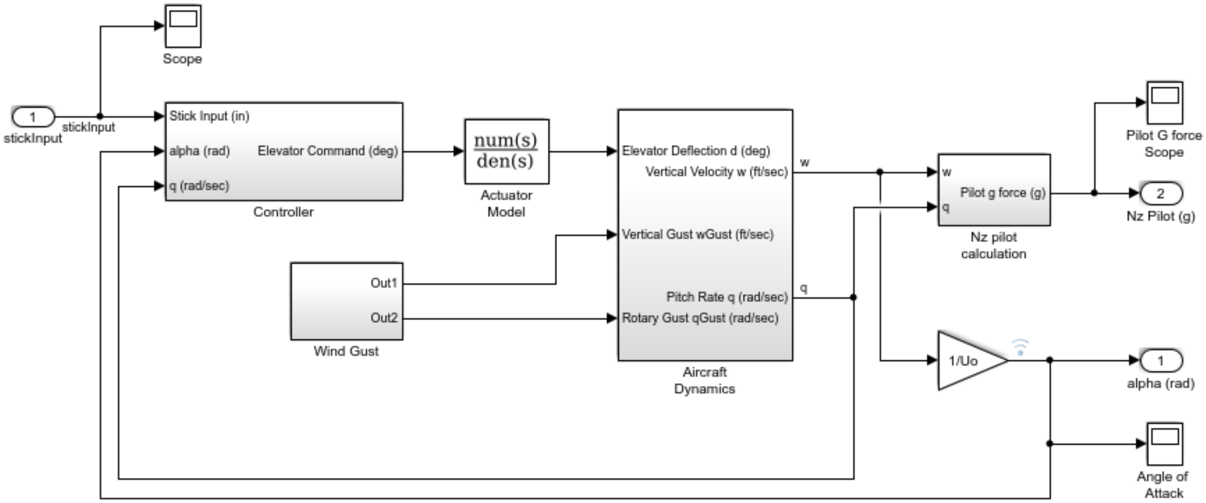
This example shows how to use MATLAB® Unit Test to test a model, and use the TAPPlugin to create TAP results. You can use TAP with CI systems. The model is a controller-plant system of a flight controller, aircraft model, and environment model.

Create a test suite and a test runner, and customize the runner with the plugin that creates the TAP file. When you run the test, it fails on several iterations. The results are written to the TAP file.

Before performing this example, set the working directory to a writable location on the path.

1. Open the Model

```
open_system(fullfile(matlabroot,'examples','simulinktest',...
    'sltestF14ParameterSweep.slx'))
```



F14 Flight Control

Copyright 1990-2016 The MathWorks, Inc.

2. Open the Test File

The test case creates a square wave input to the controller, and sweeps through 25 iterations of the parameters `a` and `b`. It compares the `alpha` output to a baseline with a tolerance of `0.0046` and fails any output which exceeds this tolerance.

```
sltest.testmanager.view;
sltest.testmanager.load(fullfile(matlabroot,'examples','simulinktest',...
    'f14ParameterSweepTest.mldatx'));
```

3. Import the `TestRunner`, `TestSuite`, `TAPPlugin`, and `ToFile` classes.

```
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import matlab.unittest.plugins.TAPPlugin
import matlab.unittest.plugins.ToFile
```

4. Create the test suite object.

```
suite = testsuite(fullfile(matlabroot,'examples','simulinktest',...
    'f14ParameterSweepTest.mldatx'))
```

```
suite = 

  1×25 Test array with properties:

    Name
    BaseFolder
    ProcedureName
    SharedTestFixtures
    Parameterization
    Tags

Tests Include:
   25 Unique Parameterizations, 0 Shared Test Fixture Classes, 0 Tags.
```

5. Create the test runner object, and set it to display output to the command window.

```
f14runner = TestRunner.withTextOutput;
```

6. Create a TAP plugin that sends output to the file `F14TapOutput.tap`.

```
tapFile = 'F14TapOutput.tap';
plugin = TAPPlugin.producingVersion13(ToFile(tapFile));
```

7. Add the plugin to the test runner.

```
addPlugin(f14runner,plugin)
```

8. Run the test. The test fails several iterations in which the delta between the signal output and the baseline exceeds the tolerance.

```
result = run(f14runner,suite);
```

```
Running f14ParameterSweepTest > New Test Suite 1
..........
........
================================================================================
Verification failed in f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sw

    ---------------------
    Framework Diagnostic:
    ---------------------
    Failed criteria: Baseline
    --> Logs:
            Inputs may not be compatible for simulation. Test results might not be accu
================================================================================
..
...
================================================================================
Verification failed in f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sw

    ---------------------
    Framework Diagnostic:
    ---------------------
    Failed criteria: Baseline
    --> Logs:
            Inputs may not be compatible for simulation. Test results might not be accu
================================================================================
.
================================================================================
Verification failed in f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sw

    ---------------------
    Framework Diagnostic:
    ---------------------
    Failed criteria: Baseline
    --> Logs:
            Inputs may not be compatible for simulation. Test results might not be accu
================================================================================
.
Done f14ParameterSweepTest > New Test Suite 1
__________

Failure Summary:

     Name
    ==================================================================================
     f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIterat
    ----------------------------------------------------------------------------------
     f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIterat
    ----------------------------------------------------------------------------------
     f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIterat
```

9. Display the results from the TAP file.

```
disp(fileread(tapFile))
```

```
TAP version 13
1..25
ok 1 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 2 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 3 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 4 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 5 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 6 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 7 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 8 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 9 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIter
ok 10 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 11 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 12 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 13 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 14 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 15 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 16 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 17 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 18 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
not ok 19 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(Scripte
    ---
    Event:
        Event Name: 'VerificationFailed'
        Event Location: 'f14ParameterSweepTest > New Test Suite 1/Iterations Parameter
        Framework Diagnostic: |
            Failed criteria: Baseline
            --> Logs:
                    Inputs may not be compatible for simulation. Test results might not
    ...
ok 20 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 21 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 22 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
ok 23 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(ScriptedIte
not ok 24 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(Scripte
    ---
    Event:
        Event Name: 'VerificationFailed'
        Event Location: 'f14ParameterSweepTest > New Test Suite 1/Iterations Parameter
        Framework Diagnostic: |
            Failed criteria: Baseline
            --> Logs:
                    Inputs may not be compatible for simulation. Test results might not
    ...
not ok 25 - f14ParameterSweepTest > New Test Suite 1/Iterations Parameter Sweep(Scripte
    ---
    Event:
        Event Name: 'VerificationFailed'
        Event Location: 'f14ParameterSweepTest > New Test Suite 1/Iterations Parameter
        Framework Diagnostic: |
            Failed criteria: Baseline
            --> Logs:
                    Inputs may not be compatible for simulation. Test results might not
    ...
```

```
sltest.testmanager.clearResults
sltest.testmanager.clear
sltest.testmanager.close

close_system('sltestF14ParameterSweep',0)
```

See Also

matlab.unittest.plugins.TAPPlugin | Test | TestResult | TestRunner | TestSuite

Related Examples

- “Run Tests for Various Workflows” (MATLAB)

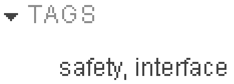
Filter Test Execution and Results

In this section...

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

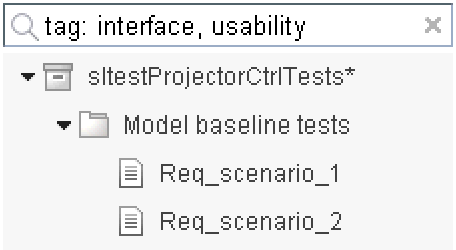
Add comma-separated tags to the **Tags** section in the Test Browser. Tags cannot contain spaces; spaces are corrected to commas.



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 `tag: id1, id2, ...` where `id1` and `id2` are example test tags. Entering multiple tags returns tests that contain any of the tags.



tag: interface, usability

sltestProjectorCtrlTests*

Model baseline tests

Req_scenario_1

Req_scenario_2

Run Filtered Tests

To run a subset of tests

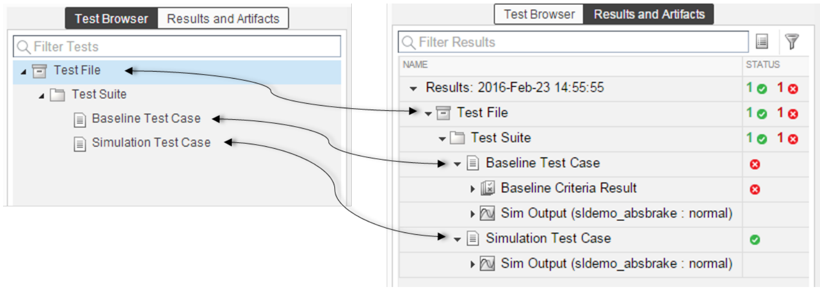
1 Filter the tests using tags.
2 In the toolstrip, click the down arrow below **Run** and select **Run Filtered**.

Test Manager Results and Reports

View Test Case Results

In this section...
"View Results Summary" on page 7-2
"Visualize Test Case Simulation Output and Criteria" on page 7-4

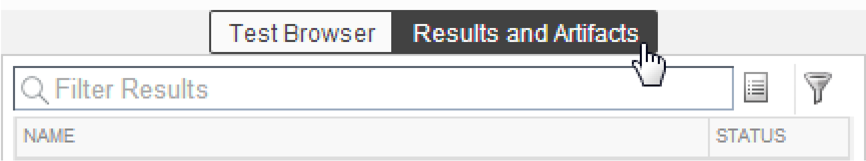
After a test case has finished running in the Test Manager, the test case result becomes available in the **Results and Artifacts** pane. Test results are organized in the same hierarchy as the test file, test suite, and test cases that were run from the **Test Browser** pane. In addition, the **Results and Artifacts** pane shows the criteria results and simulation output, if applicable to the test case.



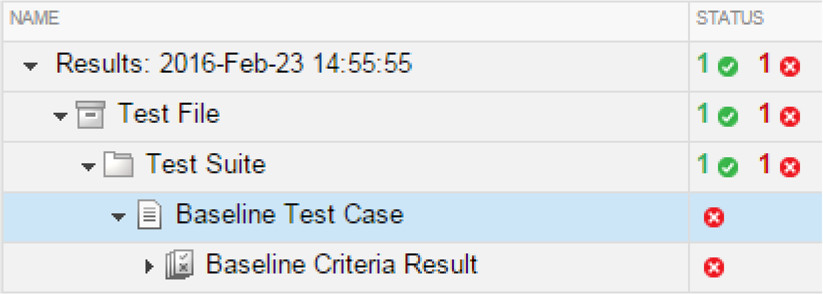
View Results Summary

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.

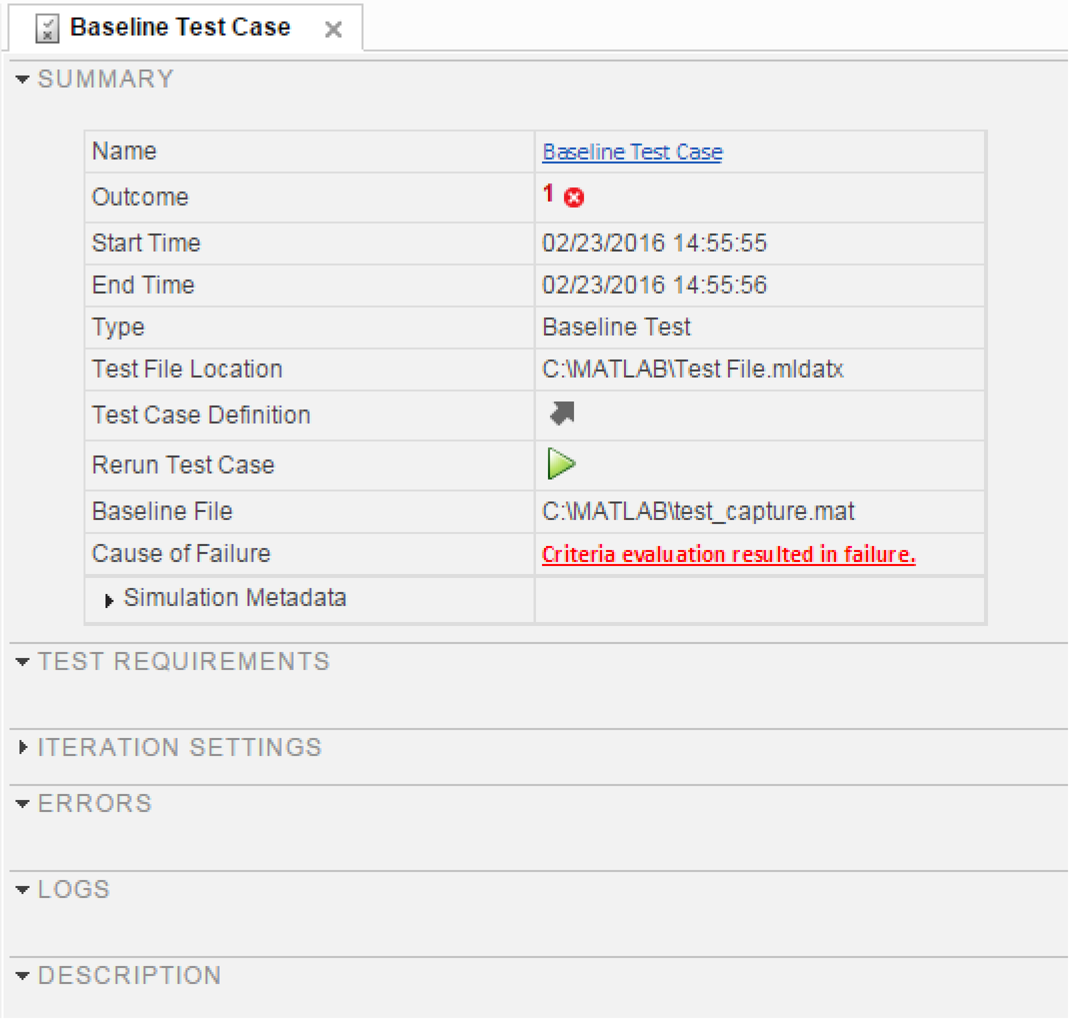


2 Double-click a test case result.



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

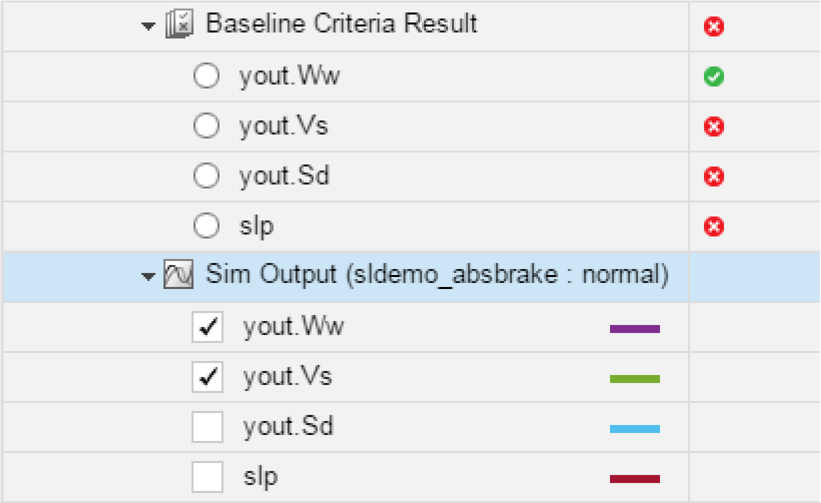
You can view signal data from simulation output or comparisons of signal data used in baseline or equivalence criteria.

To view simulation output from a test case:

1 Select the **Results and Artifacts** pane.

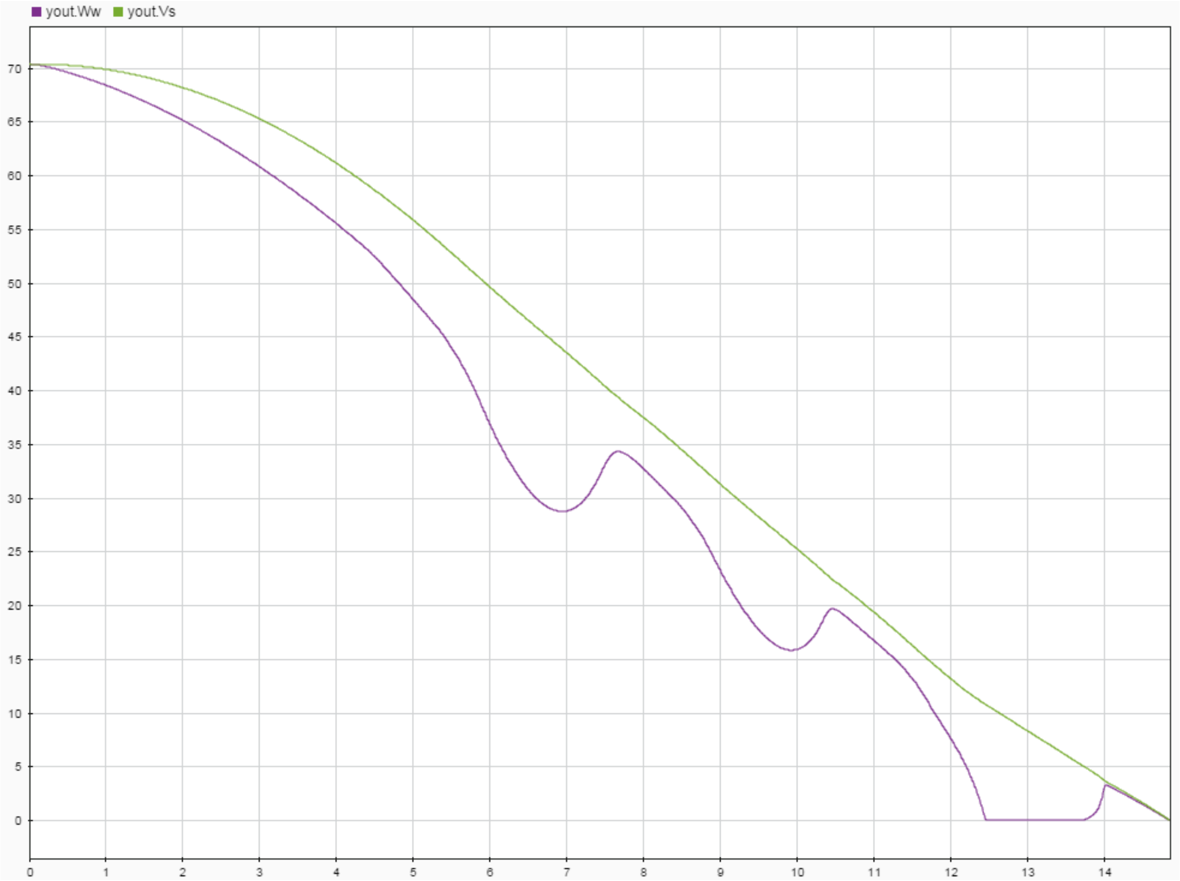
2 Expand the **Sim Output** section of the test case result.

3 Select the check box of signals you want to plot.



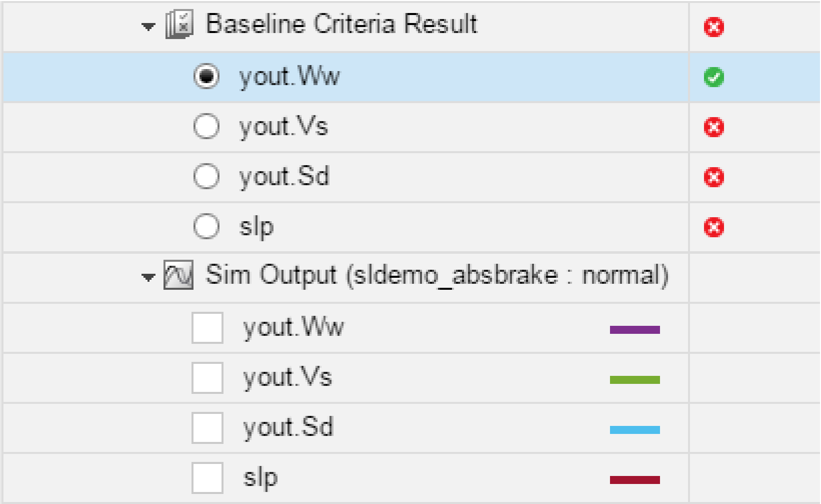
Baseline Criteria Result	✗
○ yout.Ww	✓
○ yout.Vs	✗
○ yout.Sd	✗
○ slp	✗
Sim Output (sldemo_absbrake : normal)	
☑ yout.Ww	
☑ yout.Vs	
□ yout.Sd	
□ slp	

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)

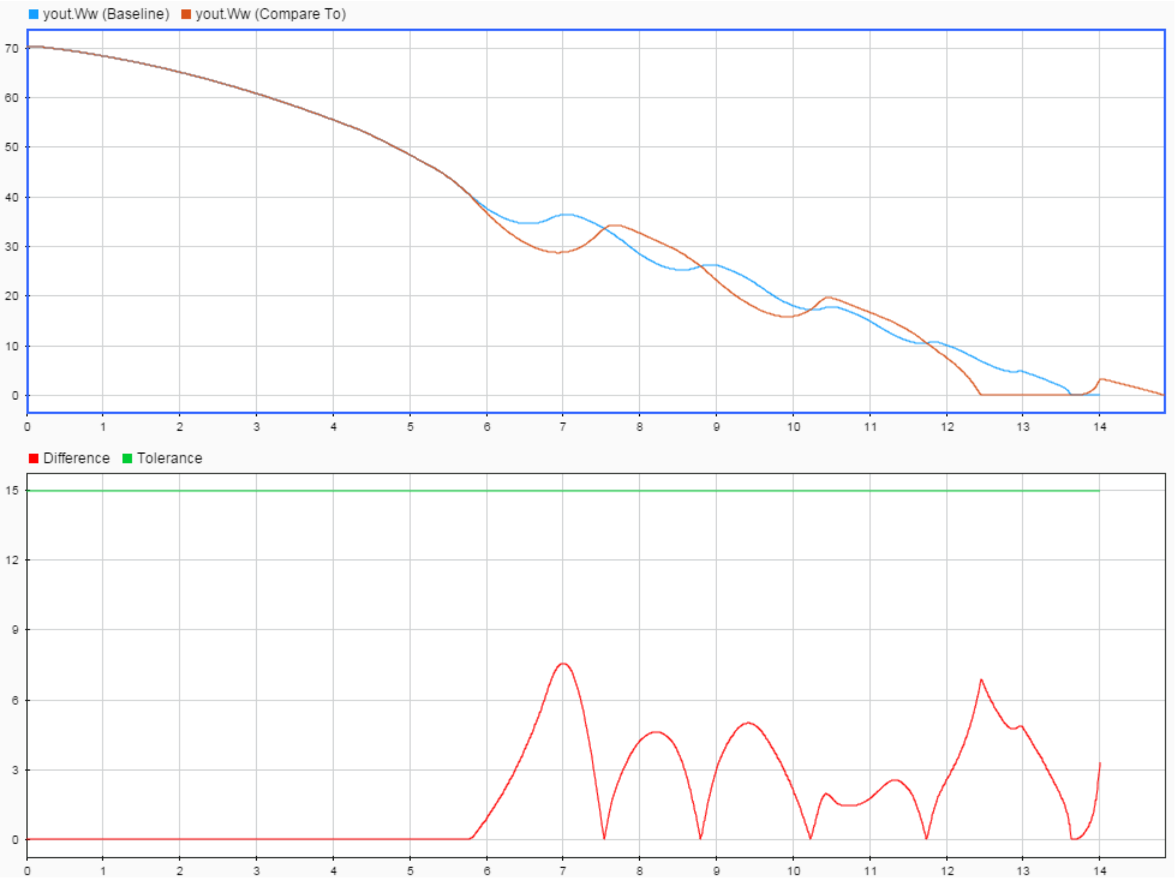
□ yout.Ww 🟪

□ 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 “Test Model Output Against a Baseline” on page 6-9.

Export Test Results and Generate Reports

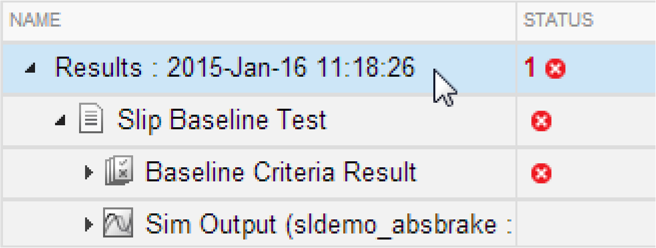
In this section...

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

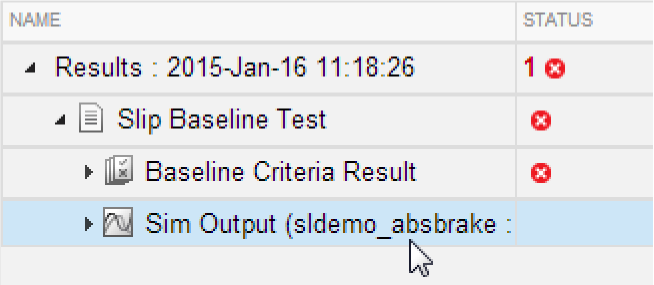
Test results are saved separately from the test file. To save results, select the result in the **Results and Artifacts** pane, and click **Export** on the toolstrip.

- Select complete result sets to export to a MATLAB data export file (`.mldatx`).



NAME	STATUS
Results : 2015-Jan-16 11:18:26	1 ⊗
Slip Baseline Test	⊗
Baseline Criteria Result	⊗
Sim Output (sldemo_absbrake :	

- 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 as needed. To include figures generated by callbacks or custom criteria, select **MATLAB figures**. For more information, see “Create, Store, and Open MATLAB Figures” on page 6-85.

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:

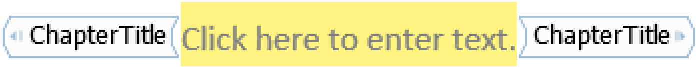
```
cd(matlabroot);
cd('help\toolbox\sltest\examples');
```

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:

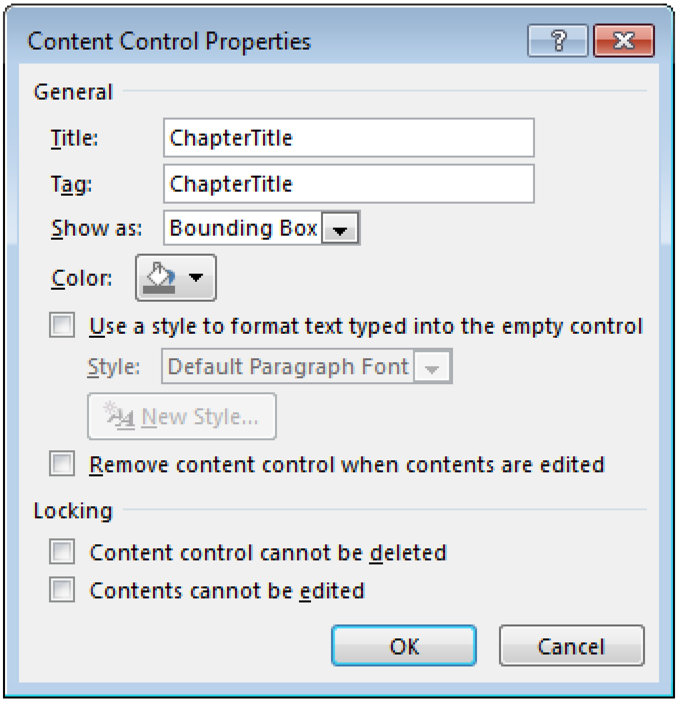
- `ChapterTitle` — report title
- `ChapterTestPlatform` — version of MATLAB used to execute tests
- `ChapterTOC` — test results table of contents
- `ChapterBody` — test results

For example, the chapter title rich text content control appears in the Microsoft Word template as:



ChapterTitle Click here to enter text. ChapterTitle

To change the control name, right-click the rich text content control and select **Properties**. Specify the control name, `ChapterTitle` or any other name, in the **Title** and **Tag** field.



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

□ 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 “Create Report Templates” (MATLAB Report Generator).
7. Click **Create**.

Related Examples

- “Create Report Templates” (MATLAB Report Generator)
- “Create, Store, and Open MATLAB Figures” on page 6-85

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.

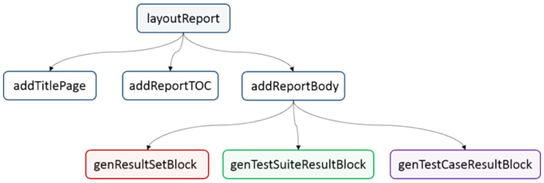
Inherit the Report Class

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 “Subclass Constructors” (MATLAB). Then, add code to the inherited class or methods to create your customizations.

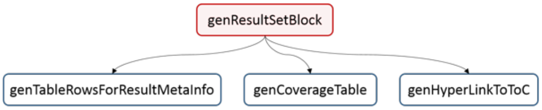
```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    %
    % Report customization code here
    %
end
```

Method Hierarchy

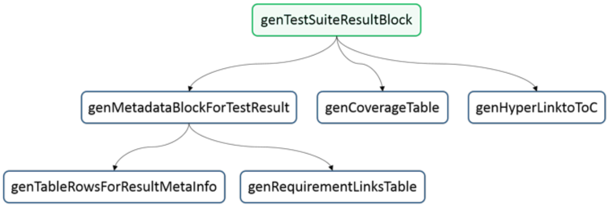
When you create the subclass, the derived class inherits methods from the `sltest.testmanager.TestResultReport` class. The body of the report is separated into three main groups: the result set block, the test suite result block, and the test case result block.



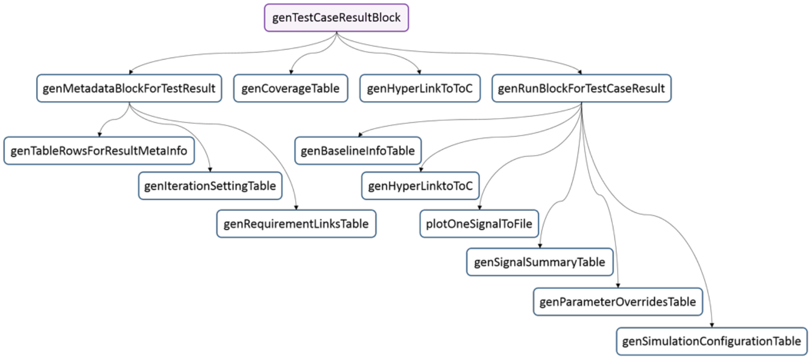
The result set block contains the result set table, the coverage table, and links to the table of contents.



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 Superclass 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 there
        % was no model used
        testSimMetaData = tcrObj.SimulationMetaData;

        if (~isempty(testSimMetaData))
            modelName = testSimMetaData.modelName;
        end

        % Get all 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);
                snapObj = SLPrint.Snapshot;
                snapObj.Target = modelName;
                snapObj.Format = 'png';
```

```
                        snapObj.FileName = fullfile(tempdir,modelName);
                        if exist(snapObj.FileName, 'file')
                            delete( snapObj.FileName );
                        end
                        snapObj.snap;
                        outputFileName = snapObj.FileName;
                        outputFileName =  [outputFileName '.png'];
                        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

- “Subclass Constructors” (MATLAB)

Append Code to a Test Report

This example shows how to use a customization class to include code in your test report. When testing systems that include handwritten code, reviewing the code itself can be part of reviewing the test results. Including the code in the test report allows you use a single document.

The example model includes handwritten C code using an S-Function builder block. The block is a component of a cruise control system; functionally, it disregards simultaneous pressing of the Accel/Res switch and the Coast/Set switch.

This example requires Simulink® Report Generator™ and Microsoft® Windows.

Navigate to the Example Folder

Before running this example, navigate to the example folder and set the filenames.

```
cd(fullfile(matlabroot,'examples','simulinktest'))

className = 'textAppendReport';
resultsFile = 'DoublePressSfcnSimTestResults';
filePath = 'textAppendedReport.zip';
```

Report Customization Class

The report custimzation class `textAppendReport.m` appends the S-Function wrapper code to the end of the report body.

```
open(className)
```

Load the Test Results and Create the Test Report

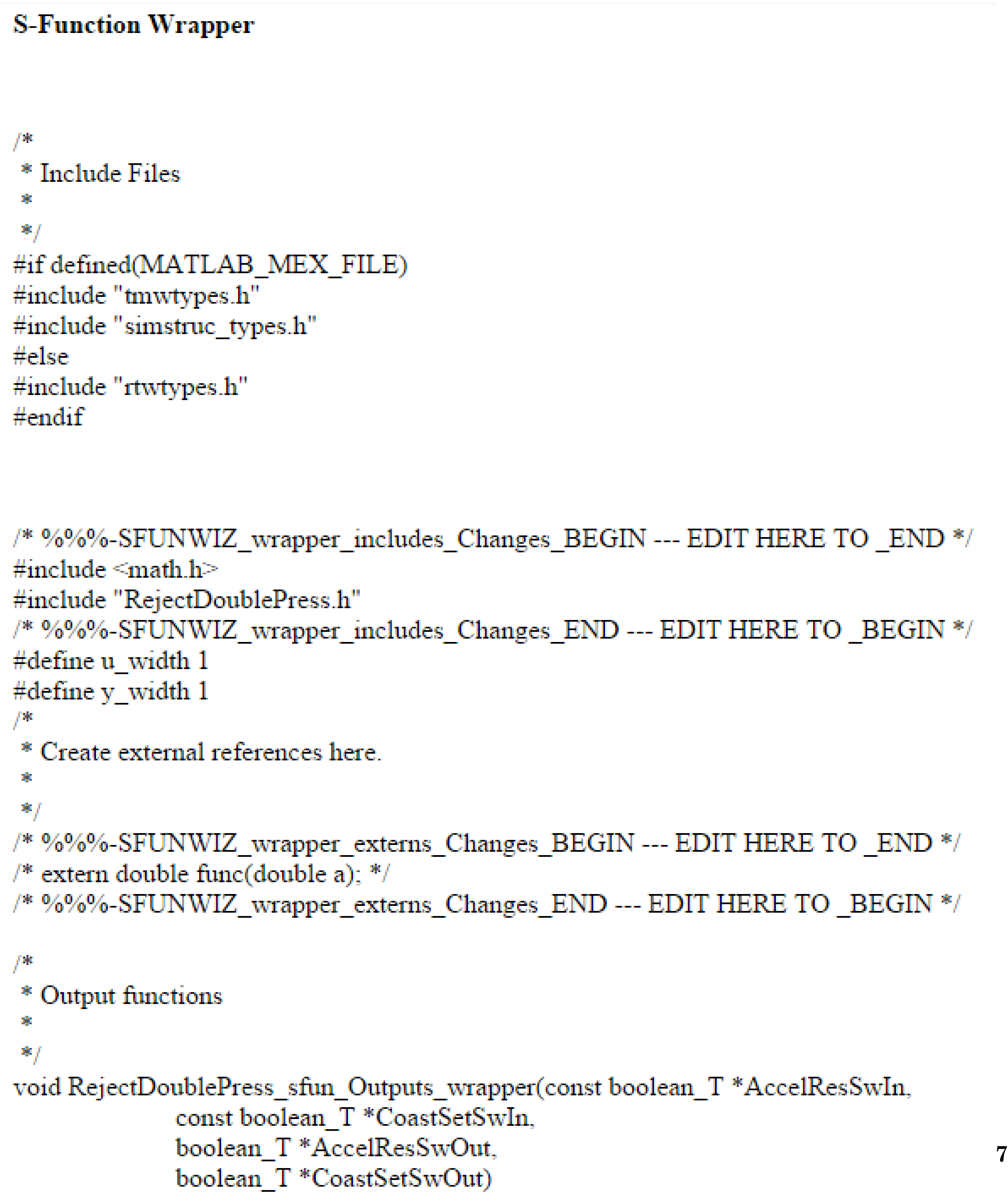
1. Load the test results file.

```
result = sltest.testmanager.importResults(resultsFile);
```

2. Create the test report using the customization.

```
sltest.testmanager.report(result,filePath,'CustomReportClass',className,...
 '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)
```

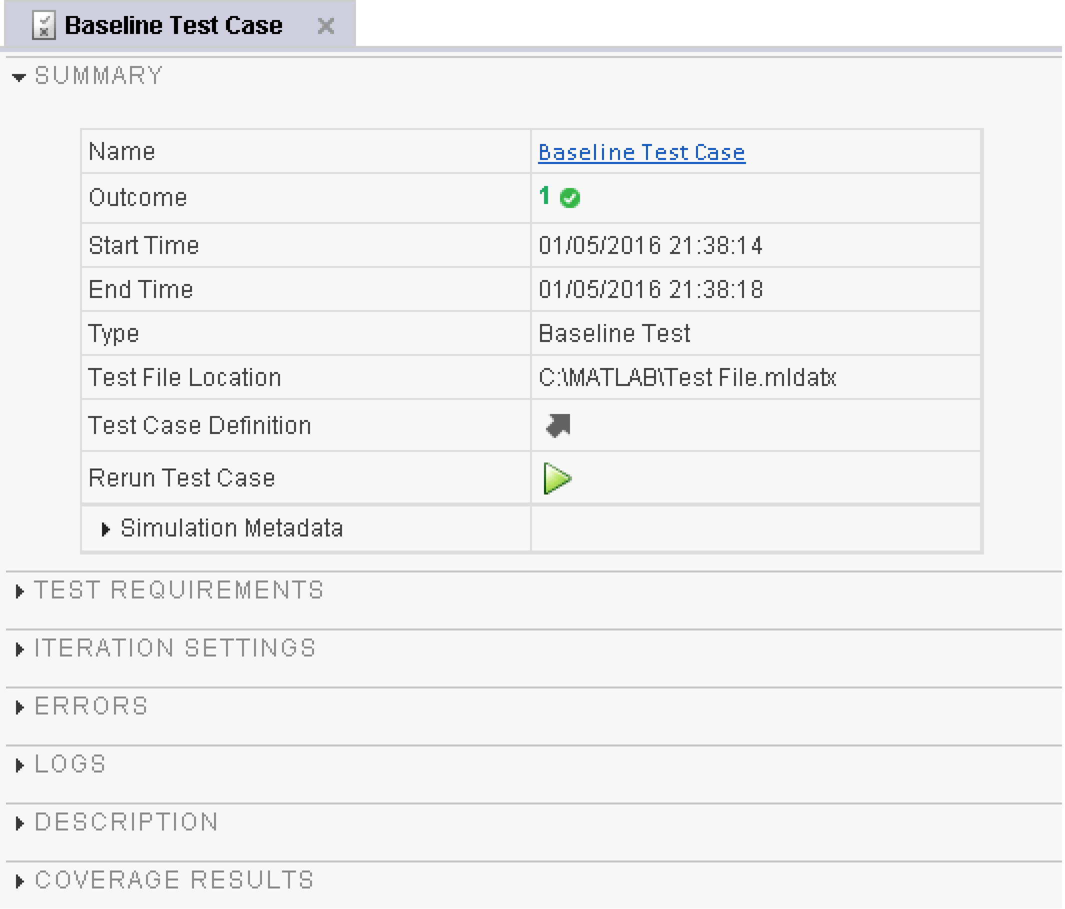
For more information on report customization, see Customize Generated Reports

```
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 all the test case result sections. A baseline test case result is shown as an example.



Baseline Test Case ×

▾ SUMMARY

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	

▸ TEST REQUIREMENTS

▸ ITERATION SETTINGS

▸ ERRORS

▸ LOGS

▸ DESCRIPTION

▸ COVERAGE RESULTS

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 any test requirements linked to the test case. See “Requirements” on page 6-60 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 “Run Multiple Combinations of Tests Using Iterations” on page 6-33.

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 any notes about the test results here. These notes are saved with the results.

Parameter Overrides

A list of any parameter overrides specified in the test case under **Parameter Overrides**. If there are no parameter overrides 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 6-43.

8

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

- If real-time test data returned from the target computer is shifted in time or is missing data points, baseline or equivalence results can consequently display a test failure. 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 3-45.
- 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:

- “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:

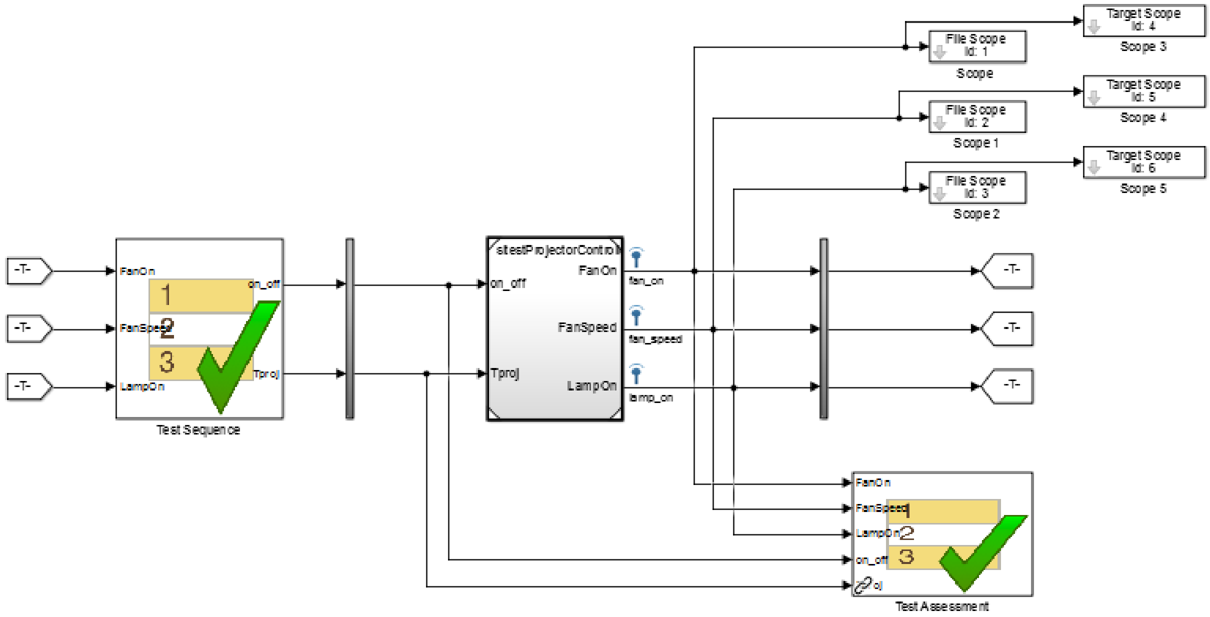
1. Open the configuration parameters.
2. Select the **All Parameters** tab and the `Data Import/Export` pane.
3. Select the **Format**.

Log Signals from Real-Time Execution

To configure your signals of interest for real-time testing:

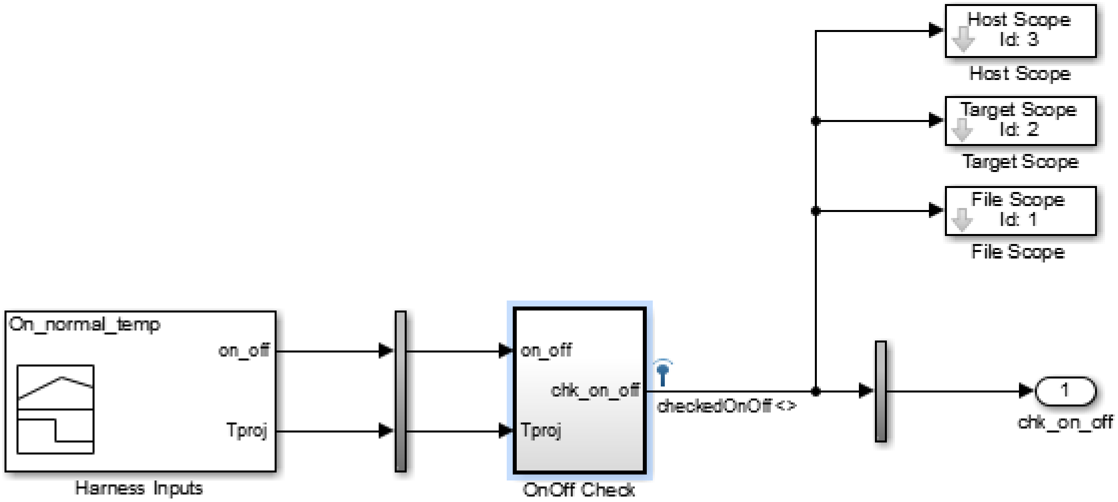
- Enable signal logging in the Configuration Parameters, in the Data Import/Export pane.
- Connect signals to Scope blocks from the Simulink Real-Time block library. Set the **Scope type** property to `File`.
- Name each signal of interest using the signal properties.

Signal naming is particularly important if you perform baseline or equivalence testing, because unnamed signals can be assigned a default name, which likely does not match the name of the baseline or equivalence signal. This test harness demonstrates four signals configured for real-time testing, using file scopes to return signal data to the Test Manager, and target scopes to display data on the target computer during execution.



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 (Simulink Real-Time).



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 a DLM file 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
Executes post-load callback	Yes	No	No

Test Case Execution Step (first to last)	Load Application From		
	Model	Target Application	Target Computer
Sets Signal Builder group	Yes	No	No
Builds DLM from model	Yes	No	No
Downloads DLM 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

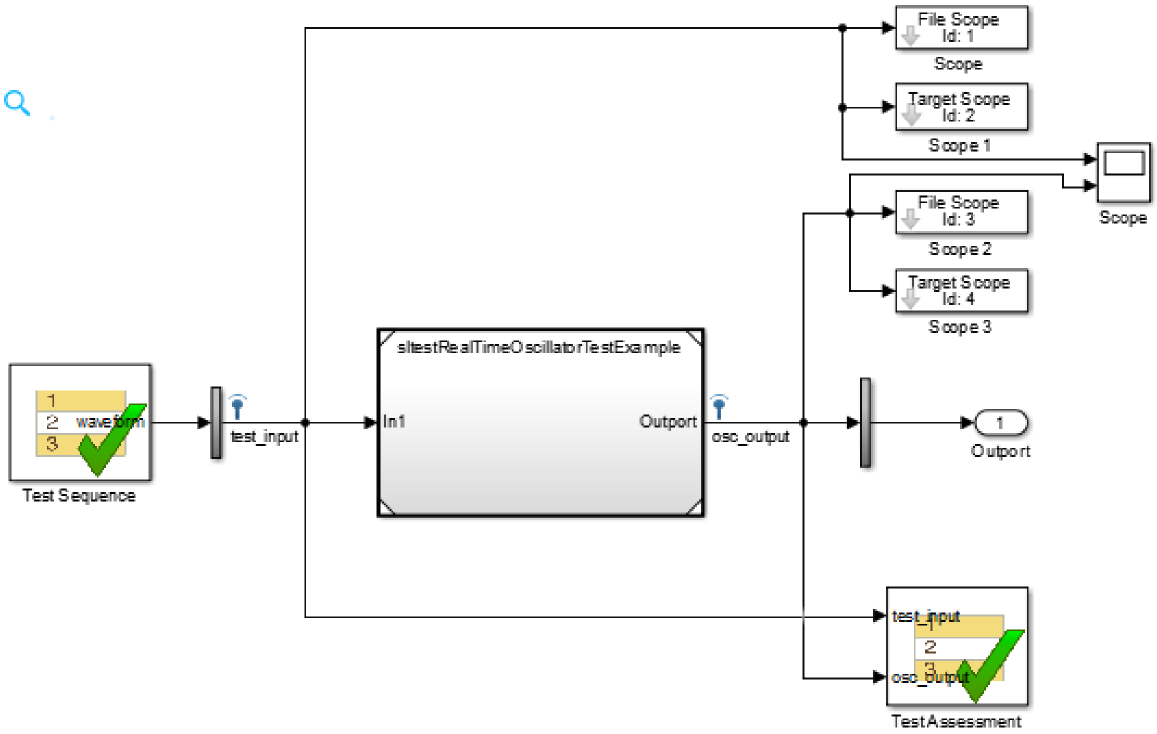
Real-time tests offer a **Pre-start real-time application** callback which executes commands just before the application executes on the target computer. Real-time test callbacks execute in a sequence along with the model load, build, download, and execute steps. Callbacks and step execution depends on how the test case loads the application.

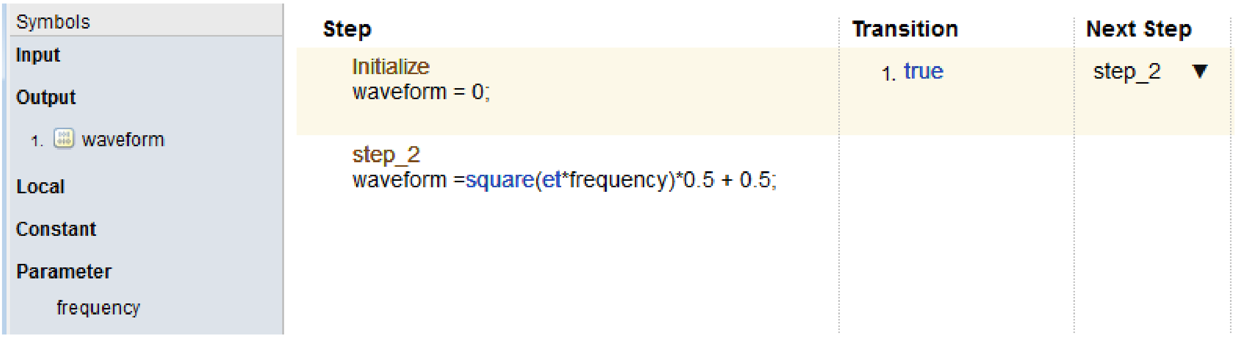
Sequence	Load application from: Model	Load application from: Target application	Load application from: Target computer
Executes first	**Preload callback**	**Preload callback**	**Preload callback**
	Post-load callback	—	—
	Pre-start real-time callback	**Pre-start real-time callback**	**Pre-start real-time 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 “Run Multiple Combinations of Tests Using Iterations” on page 6-33. 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`.





Symbols

Input

Output

1. waveform

Local

Constant

Parameter

frequency

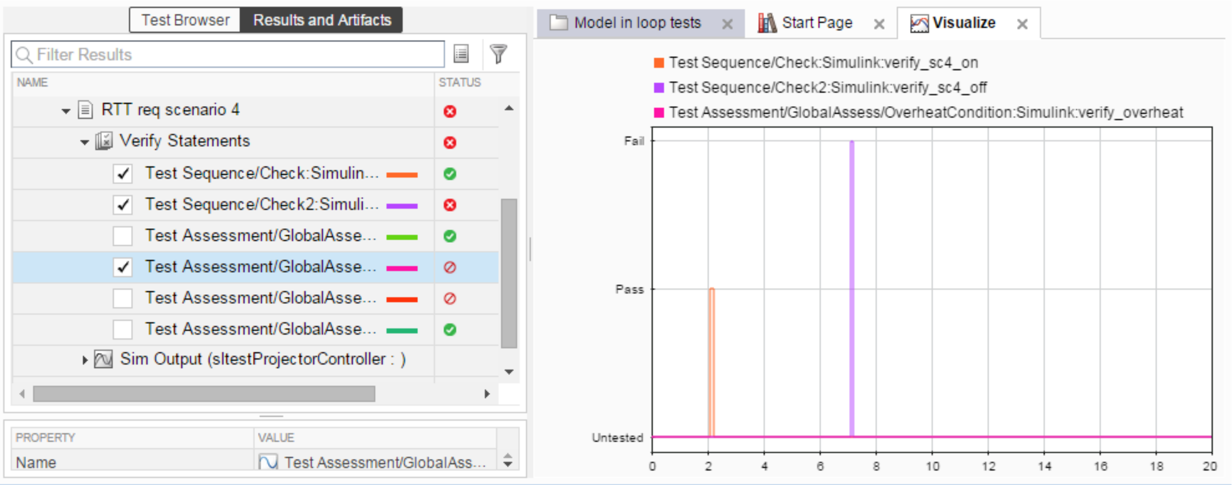
Step	Transition	Next Step
Initialize waveform = 0;	1. true	step_2 ▼
step_2 waveform =square(et*frequency)*0.5 + 0.5;		

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 Simulation Using Logical Statements” on page 3-25.



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...

- 🟧 Test Sequence/Check:Simulink:verify_sc4_on
- 🟪 Test Sequence/Check2:Simulink:verify_sc4_off
- 🟥 Test Assessment/GlobalAssess/OverheatCondition:Simulink:verify_overheat

Related Examples

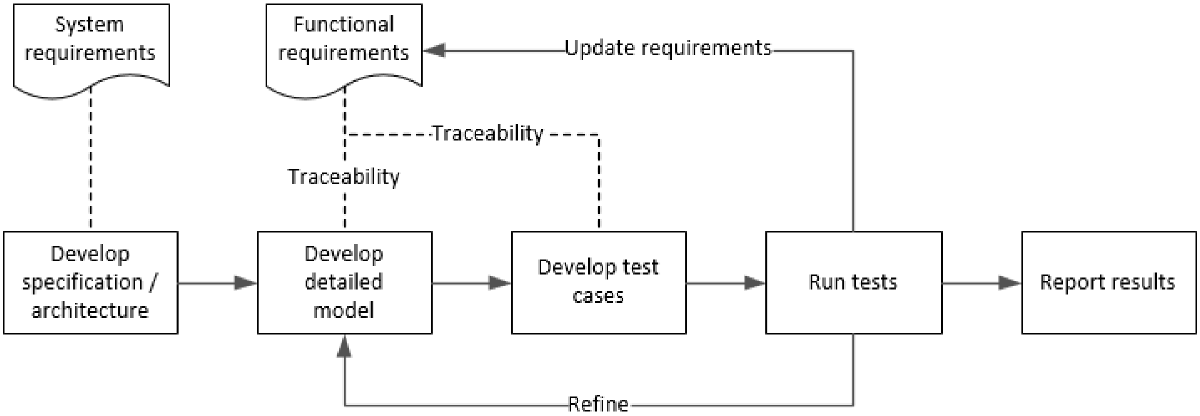
- “Test Real-Time Application” (Simulink Real-Time)

Verification and Validation

Test Model Against Requirements and Report Results

Requirements Overview

Requirements are the basis for your system architecture, algorithm, and test plan. Traceability between requirements documents, model, code, and tests helps you document relationships, manage design changes, and interpret test results. Required model properties and test objectives enable targeted design analysis and test case generation for specific scenarios. You can evaluate your design and identify incomplete or missing requirements with ad-hoc testing, using simulated user interfaces for your model. Also, you can use rapid prototyping to validate requirements, and connect to physical or simulated environments to test your algorithm. Update the design, adding requirements and requirements links as necessary.



Test a Cruise Control Safety Requirement

This example shows a requirements-based testing workflow for a cruise control model. You start with a model that has traceability to an external requirements document. You add a test to evaluate two safety requirements, checking that the cruise control disengages when the system reaches certain conditions. You add traceability to this test, run the test, and report the results.

1 Create a copy of the project in a working folder. Enter

```
slVerificationCruiseStart
```

2. Open the model and the test harness. On the command line, enter

   ```
   open_system simulinkCruiseAddReqExample
   sltest.harness.open('simulinkCruiseAddReqExample','SafetyTest_Harness1')
   ```
3. Open the Test Sequence block.

   - The `BrakeTest` sequence tests that the system disengages when the brake pedal is pressed. It includes a `verify` statement

     ```
     verify(engaged == false,...
         'verify:brake',...
         'system must disengage when brake applied')
     ```
   - The `LimitTest` sequence tests that the system disengages when the speed exceeds a limit. It includes a `verify` statement

     ```
     verify(engaged == false,...
         'verify:limit',...
         'system must disengage when limit exceeded')
     ```
4. Open the requirements document. In the Simulink Project window, expand the **documents** folder and open **simulinkCruiseChartReqs.docx**.
5. Add links between the test steps and the requirements document.

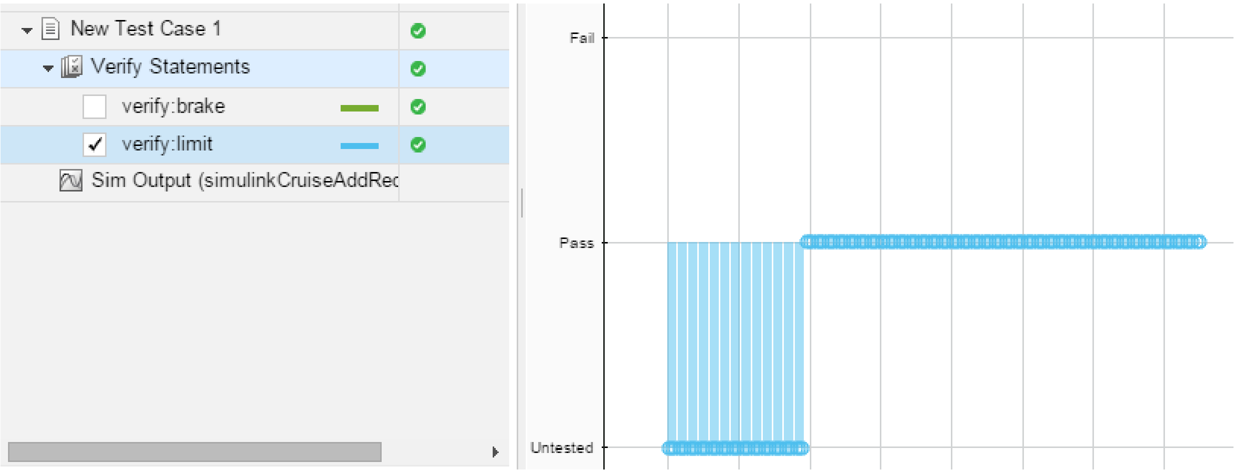
   1. In the requirements document, highlight item 3.1, “Vehicle braking will transition system to disengaged (inactive) when engaged (active)”
   2. With item 3.1 highlighted, in the test sequence, right-click the `BrakeTest` step. Select **Requirements traceability** > **Link to Selection in Word**.
   3. In the requirements document, highlight item 3.4, “Transition to disengaged (inactive) when vehicle speed is outside the limits of 20 mph to 90 mph”
   4. With item 3.4 highlighted, in the test sequence, right-click the `LimitTest` step. Select **Requirements traceability** > **Link to Selection in Word**.
   5. Save the requirements document and the model.
6. Create a test case in the Test Manager, and link the test case to the requirements section.

   1. Open the Test Manager. In the Simulink menu, select **Analysis** > **Test Manager**.
   2. In the Test Manager toolstrip, click **New** > **Test File**. Select the tests folder in the project, and enter a name for the test file. Click **Save**.

      A new baseline test is created.

    3. Under **System Under Test**, in the **Model** field, click the button to use the current model. The field displays the model name.
    4. Expand the **Test Harness** section. From the drop-down menu, select the test harness name.
    5. In the requirements document, highlight section 3.1.
    6. In the test case, expand the **Requirements** section. Click the arrow next to the **Add** button and select `Link to Selection in Word`.
    7. Use the same process to link the test case to section 3.4 in the requirements document.

7. Highlight the test case. In the Test Manager toolstrip, click **Run**.
8. When the test finishes, expand the **Verify Statements** results. The results show that both assessments pass, and the plot shows the detailed results of each statement.



9. Create a report using a custom Microsoft Word template.

    1. In the Test Manager, right-click the test case name. Select **Results: > Create Report**.
    2. In the Create Test Result Report dialog box, set the options:

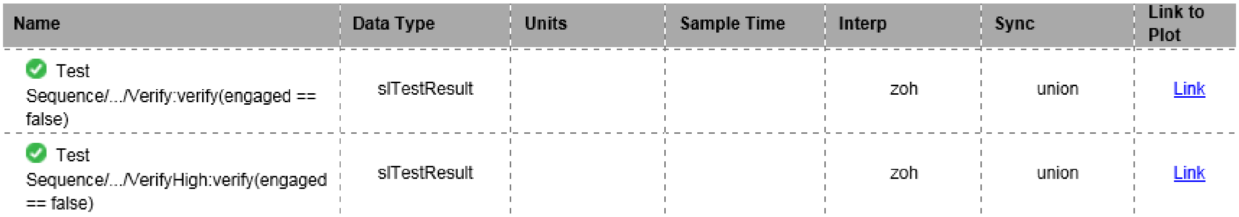
        - Title: `SafetyTest`
        - Results for: `All Tests`

   - File Format: `DOCX`
   - For the other options, keep the default selections.

   3. For the **Template File**, select the `ReportTemplate.dotx` file in the **documents** project folder.
   4. Enter a file name and select a location for the report.
   5. Click **Create**.

10. Review the report.

    1. In the **Test Case Requirements** section, click the link to trace to the requirements document.
    2. 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

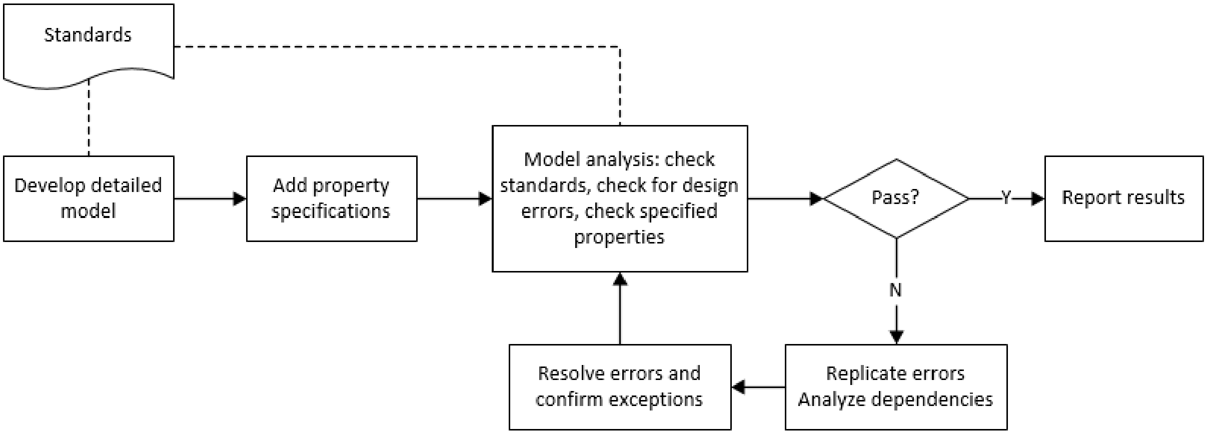
Related Examples

- "Link to Requirements Modeled in Simulink" (Simulink Verification and Validation)
- "Link Tests to Requirements" on page 1-2
- "Validate Requirements Links in a Model" (Simulink Verification and Validation)
- "Create Requirements Traceability Report for Model" (Simulink Verification and Validation)

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

   ```
   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 Verification and Validation.

   a. In the left pane, in the **By Product > Simulink Verification and Validation > 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 Verification and Validation 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**.

Related Examples

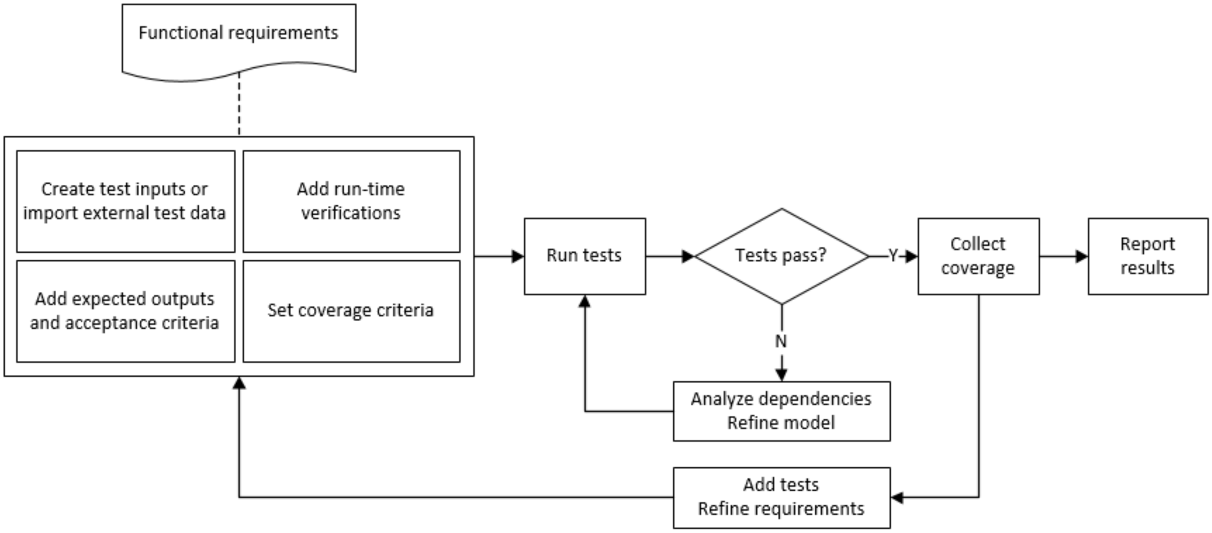
- "Check for Compliance in Model and Subsystems" (Simulink Verification and Validation)
- "Collect Model Metrics Using the Model Advisor" (Simulink Verification and Validation)
- "Run a Design Error Detection Analysis" (Simulink Design Verifier)
- "Prove Properties in a Model" (Simulink Design Verifier)

Perform Functional Testing and Analyze Test Coverage

Functional Testing and Coverage Analysis Overview

Functional testing starts 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 regularly. 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.



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 Verification and Validation, 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:

```
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 9-2. At the command line, enter:

```
open slReqTests.mldatx
```

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.

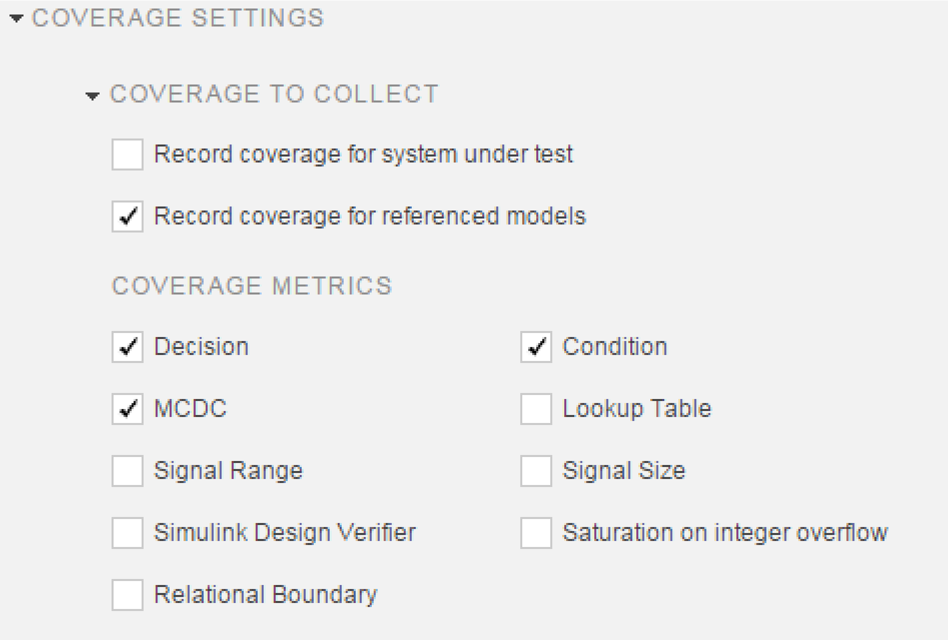
a Open the test manager. In the Simulink menu, click **Analysis** > **Test Manager**.

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**.

e Under **COVERAGE METRICS**, select **Decision**, **Condition**, and **MCDC**.



▾ COVERAGE SETTINGS

▾ COVERAGE TO COLLECT

☐ Record coverage for system under test

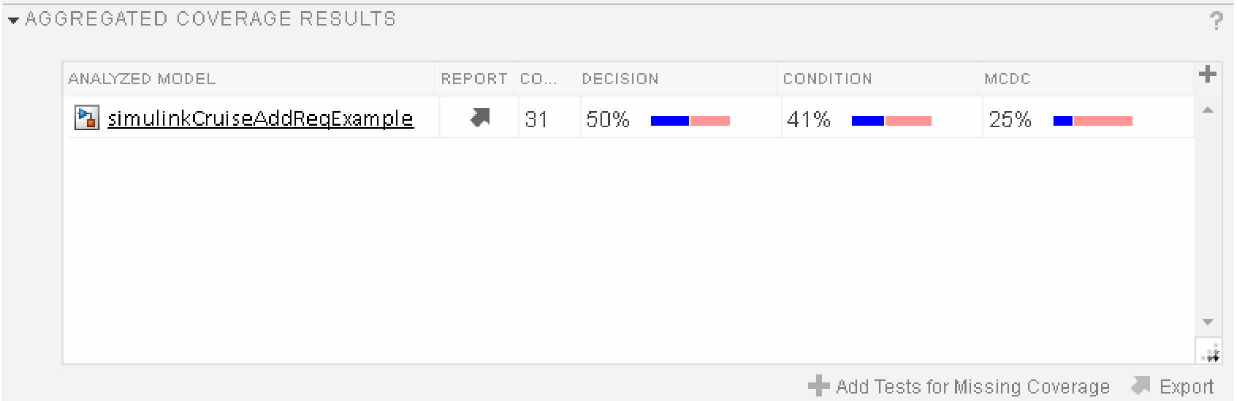
☑ Record coverage for referenced models

COVERAGE 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.



▾ AGGREGATED COVERAGE RESULTS

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.

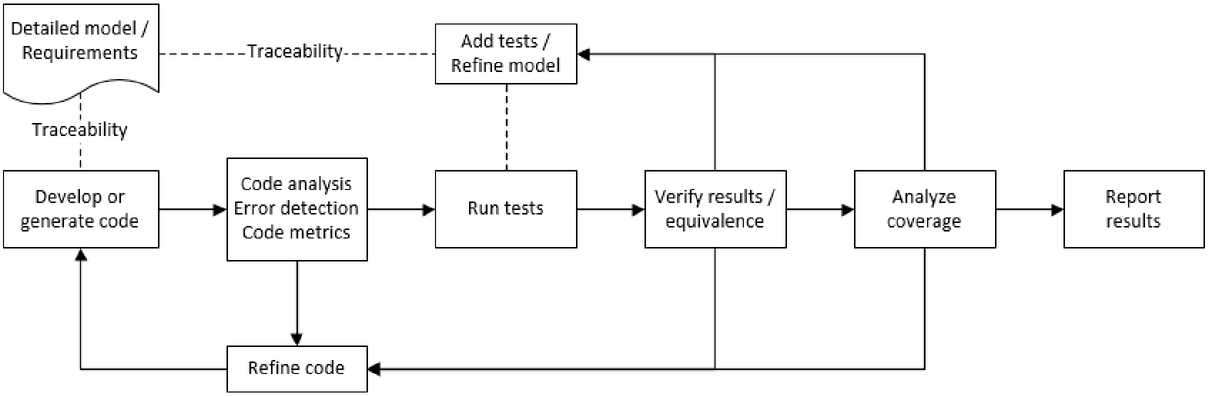
Related Examples

- "Link Tests to Requirements" on page 1-2
- "Assess Simulation Using Logical Statements" on page 3-25
- "Test Model Output Against a Baseline" on page 6-9
- "Highlight Functional Dependencies" (Simulink Design Verifier)
- "Generate Test Cases for Model Decision Coverage" (Simulink Design Verifier)
-

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.



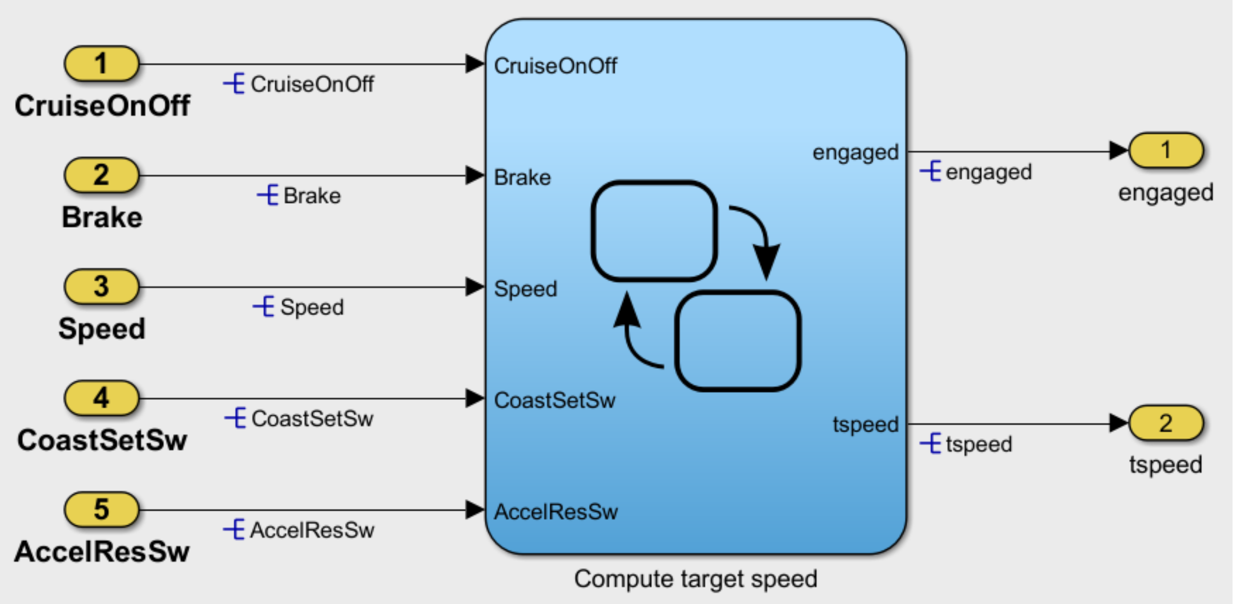
Analyze Code for Defects, Metrics, and MISRA C:2012

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 advisors. 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 Simulink project:

```
slVerificationCruiseStart
```

2 From the Simulink project, open the model `simulinkCruiseErrorAndStandardsExample`.

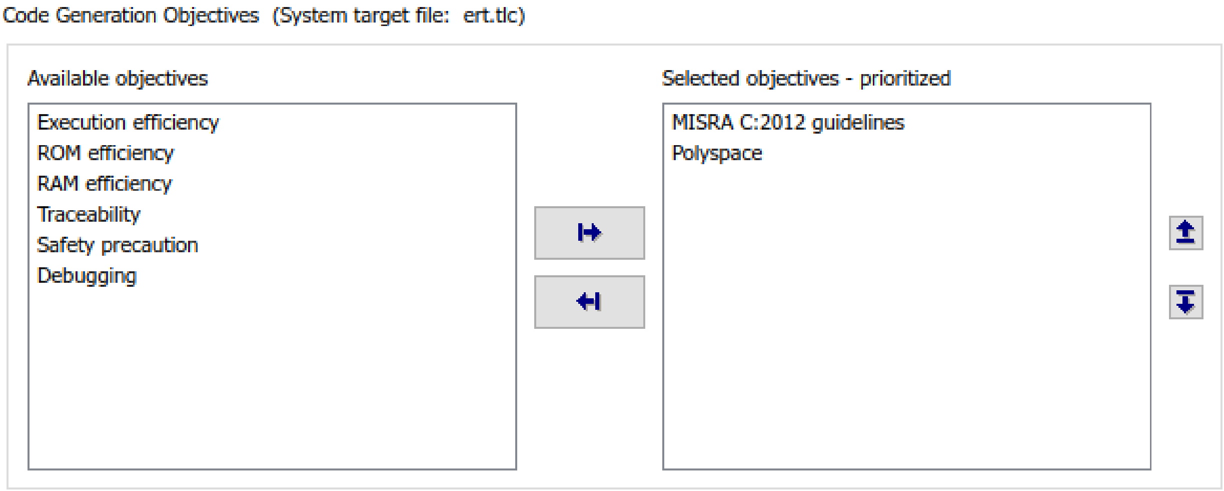


Run Code Generator Checks

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

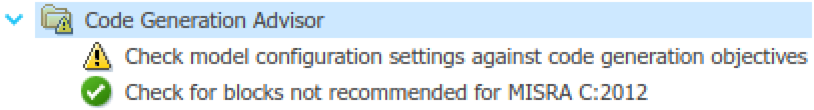
Execution efficiency
ROM efficiency
RAM efficiency
Traceability
Safety precaution
Debugging

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**.

For your own model, you might not want to use all the recommended configuration settings. Using nonrecommended settings can generate less MISRA compliant code.

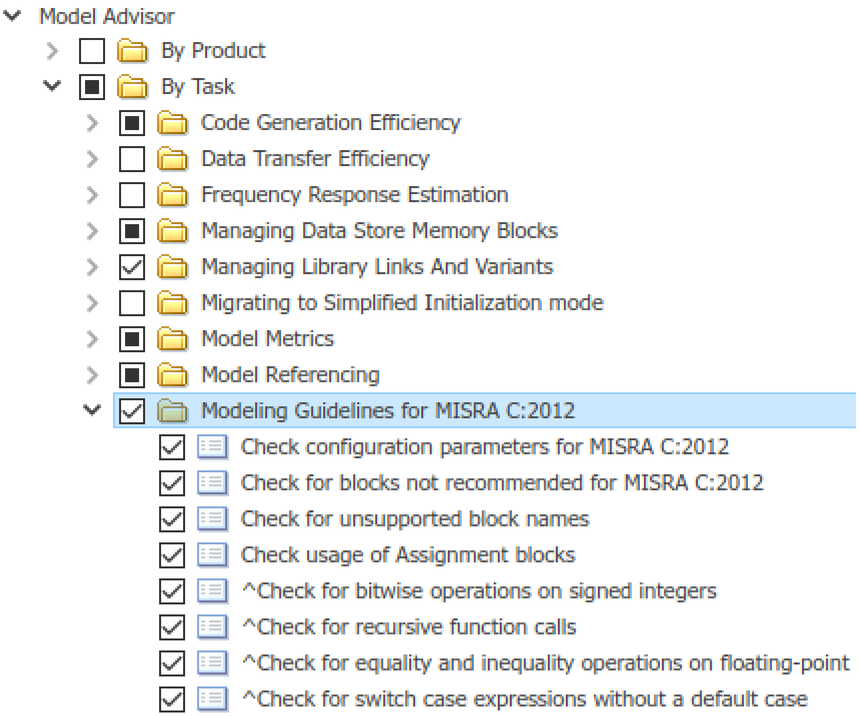
Run Model Advisor Checks

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.

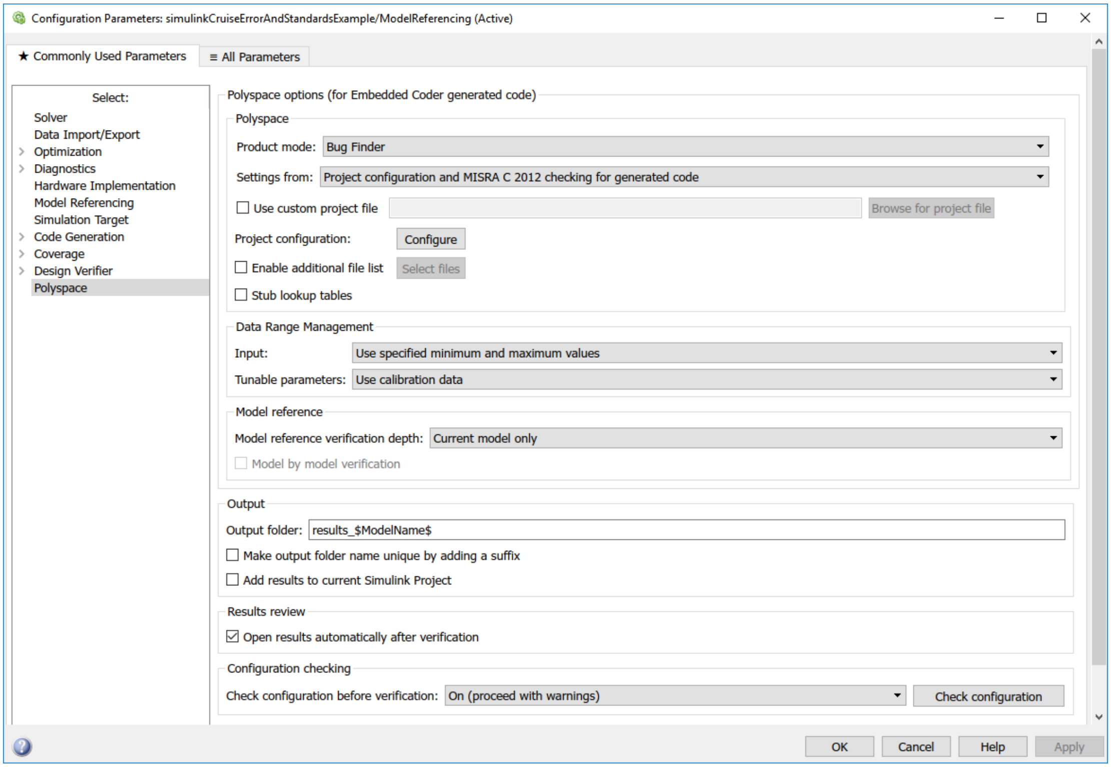
4 If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

For your own model, you might not want to use all the recommendations. Using nonrecommended settings or blocks can generate less MISRA compliant code.

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

Project configuration:

□ Enable additional file list

□ 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

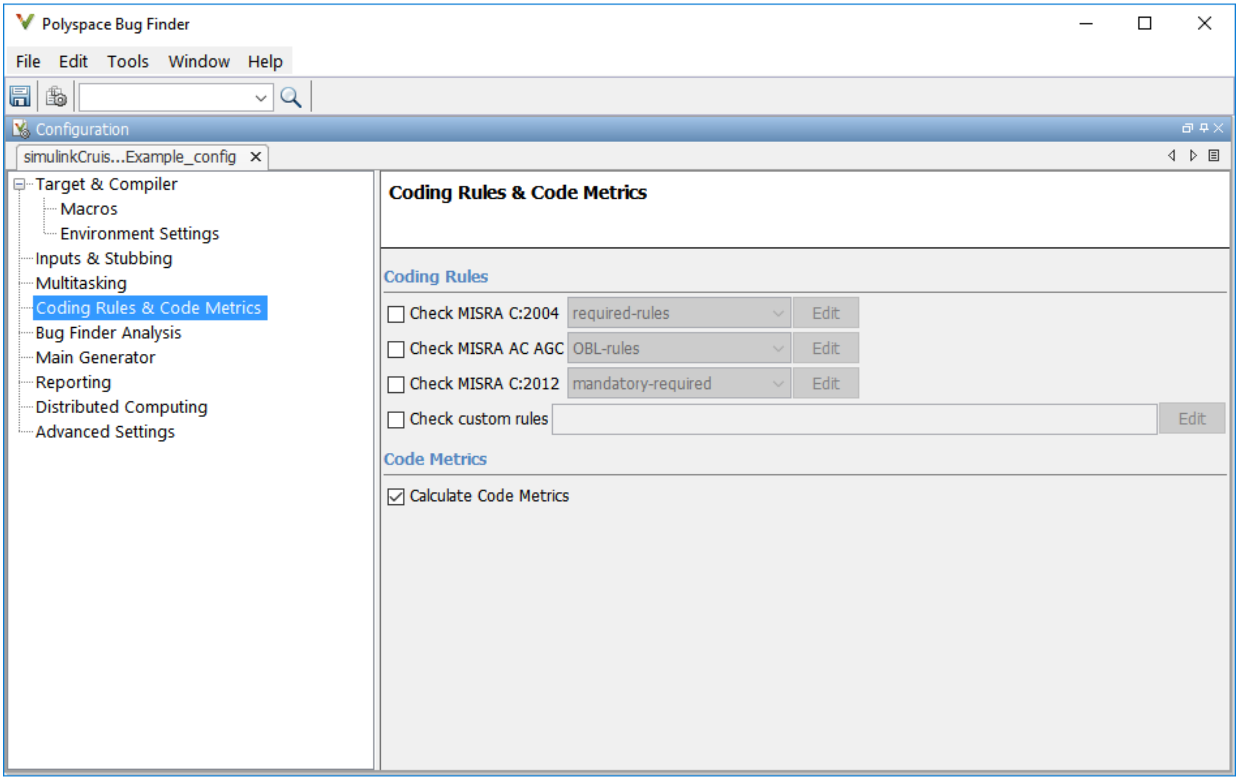
Results review

☑ Open results automatically after verification

Configuration checking

Check configuration before verification: On (proceed with warnings)

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

☑ Calculate Code Metrics

5 On the same pane, select **Calculate Code Metrics** (Polyspace Bug Finder). 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**.

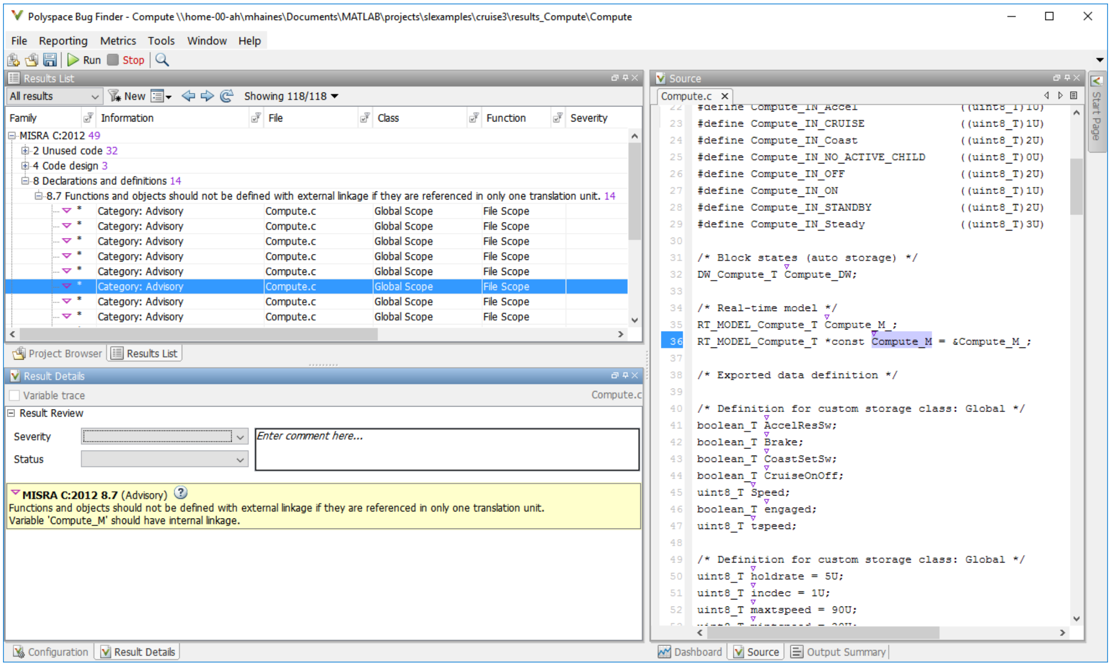
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

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.



Polyspace Bug Finder - Compute \\home-00-ah\mhaines\Documents\MATLAB\projects\slexamples\cruise3\results_Compute\Compute

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	

Severity

Status

Enter comment here...

▽ **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.

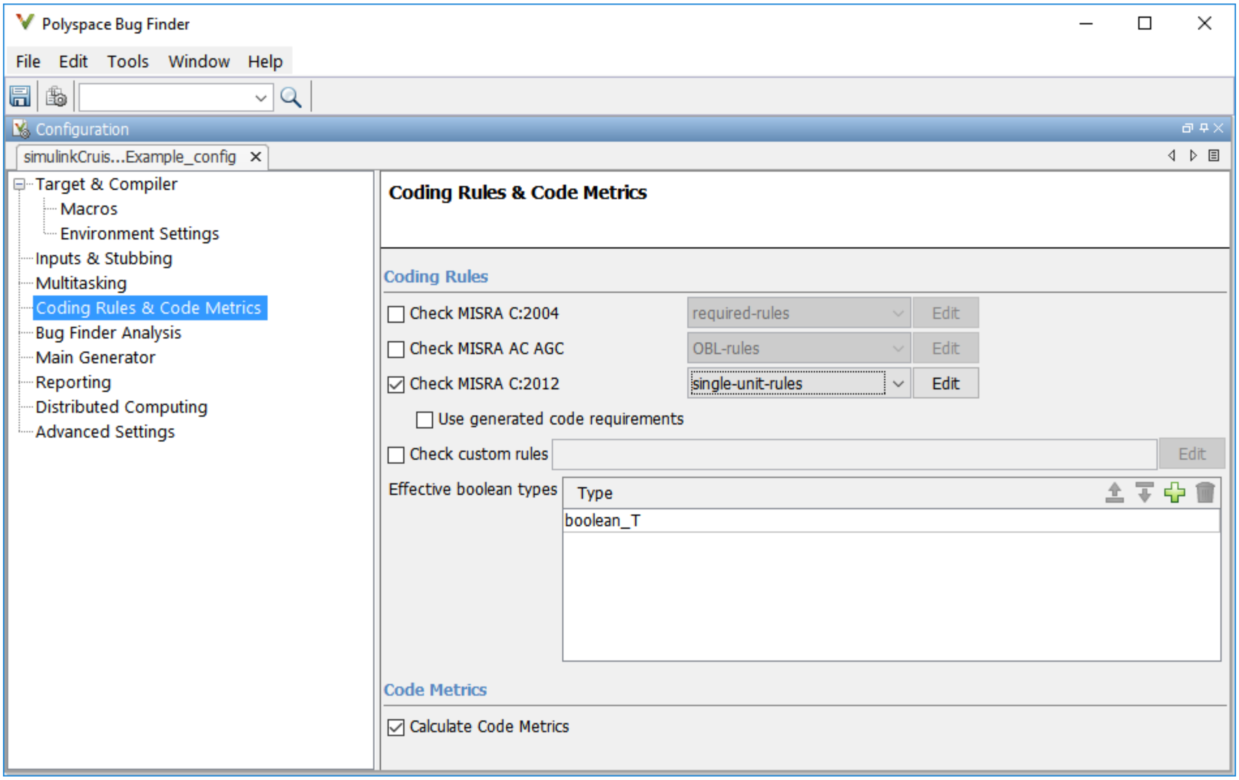
```
#define Compute_IN_Accel                  ((uint8_T)1U)
#define Compute_IN_CRUISE                 ((uint8_T)1U)
#define Compute_IN_Coast                  ((uint8_T)2U)
#define Compute_IN_NO_ACTIVE_CHILD        ((uint8_T)0U)
#define Compute_IN_OFF                    ((uint8_T)2U)
#define Compute_IN_ON                     ((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 allow 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** (Polyspace Bug Finder) 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

Type
boolean_T

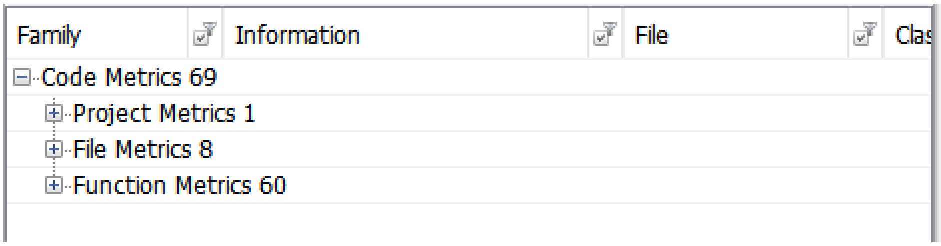
Code Metrics

☑ 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 (Polyspace Bug Finder).

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**.

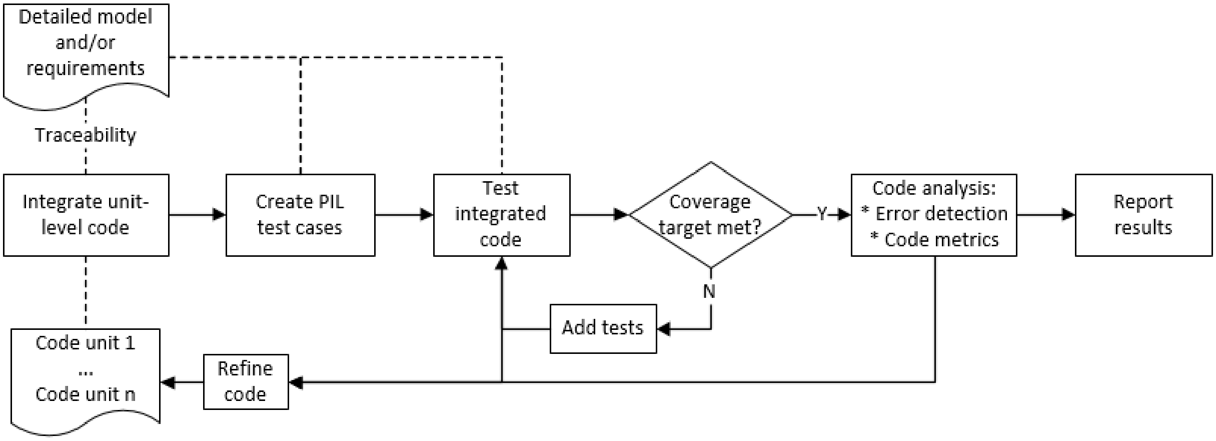
Related Examples

- “Generate and Analyze Code” (Polyspace Bug Finder)
-
- “Export Test Results and Generate Reports” on page 7-9

Module Verification and Testing Processor-in-the-Loop

Module Verification and Testing Processor-in-the-Loop Overview

Module verification involves testing and analyzing code at a system level, integrating generated code from your model, handwritten code, and legacy code. Module verification often includes generating code that executes on a target object, rather than the desktop environment. Analyze the code to resolve errors and evaluate key metrics. Test the integrated system using new requirements-based tests and system-level tests from your model. Collect coverage on these tests and add tests to meet coverage targets.



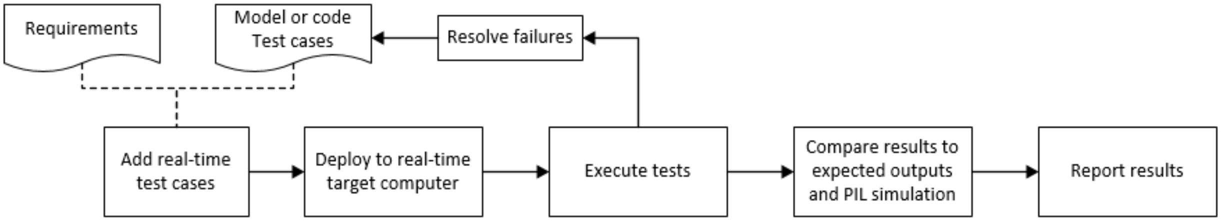
Related Examples

-
- “Generate and Analyze Code” (Polyspace Bug Finder)

Test a Model in Real Time

Real-Time Testing and Testing Production Models Overview

Real-time testing assesses the system while including the effects of timers, physical signals, and target hardware. Sweep through parameter values on the target, verify system operation during execution, and verify expected results in the desktop environment. Systems that have been verified on target hardware often exist in a change-controlled state. You can test these systems without modifying them by using isolated simulation and analysis environments.



Related Examples

- “Create and Run Real-Time Application from Simulink Model” (Simulink Real-Time)
- “Test Models in Real Time” on page 8-2
- “Assess Simulation Using Logical Statements” on page 3-25