

Simulink<sup>®</sup> Test<sup>™</sup>

User's Guide



MATLAB<sup>®</sup>&SIMULINK<sup>®</sup>

R2016a



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

*Simulink<sup>®</sup> Test<sup>™</sup> User's Guide*

© COPYRIGHT 2015–2016 by The MathWorks, Inc.

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

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

### Trademarks

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

### Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

### 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)

## Test Strategies

1

<b>Functional Testing in Verification</b> .....	1-2
<b>Link Tests to Requirements</b> .....	1-3
Requirements Traceability Considerations .....	1-3
Establish Requirements Traceability for Testing .....	1-4

## Test Harness

2

<b>Test Harness and Model Relationship</b> .....	2-2
Test Harness Description .....	2-2
Harness — Model Relationship for a Model Component .....	2-3
Harness — Model Relationship for a Top-Level Model .....	2-4
Resolving Parameters .....	2-5
<b>Considerations and Limitations</b> .....	2-6
Test Harness .....	2-6
Test Sequence Block .....	2-6
<b>Select Test Harness Properties</b> .....	2-8
Create a Test Harness .....	2-8
Considerations for Selecting Test Harness Properties .....	2-8
Save Test Harnesses Externally .....	2-9
Choosing Sources and Sinks .....	2-9
Use Separate Assessment Block .....	2-9
Initial Harness Configuration .....	2-9
Verification Modes .....	2-11
Change Harness Properties .....	2-11

<b>Test Harness Parameters and Signals</b> .....	<b>2-12</b>
Test Harness Generation Without Compilation .....	2-12
Signal Conversion Subsystem .....	2-12
<b>Refine, Test, and Debug a Subsystem</b> .....	<b>2-14</b>
Model and Requirements .....	2-14
Create a Harness for the Controller .....	2-16
Inspect and Refine the Controller .....	2-18
Add a Test Case and Test the Controller .....	2-19
Debug the Controller .....	2-20
<b>Manage Test Harnesses</b> .....	<b>2-23</b>
Internal and External Test Harnesses .....	2-23
Manage External Test Harnesses .....	2-23
Convert Between Internal and External Test Harnesses ...	2-24
Preview and Open a Test Harness .....	2-26
Find Test Cases Associated with a Test Harness .....	2-27
Export Test Harnesses to Separate Models .....	2-27
Clone and Export a Test Harness to a Separate Model ....	2-28
Delete Test Harnesses in a Model .....	2-30
<b>Synchronize Changes Between Test Harness and Model</b> ..	<b>2-32</b>
Maintain SIL or PIL Block Fidelity .....	2-32
Synchronize Changes to the Component Under Test .....	2-32
Rebuild Test Harness .....	2-33
Update Parameters from Test Harness to Model .....	2-33
<b>Test Library Blocks</b> .....	<b>2-37</b>
Library Testing Workflow .....	2-37
Library and Linked Subsystem Test Harness .....	2-38
Edit Library Block from a Test Harness .....	2-39

## Test Sequences and Assessments

# 3

<b>Introduction to Test Sequences</b> .....	<b>3-2</b>
Structure of a Test Sequence .....	3-2
Test Sequence Hierarchy .....	3-2
Step Transitions .....	3-2
Create a Basic Test Sequence .....	3-3

<b>Organize Test Sequences</b> .....	<b>3-8</b>
<b>Test Sequence Action and Transition Operations</b> .....	<b>3-11</b>
Transition Between Steps Using Temporal or Signal Conditions .....	<b>3-11</b>
Link a Test Assessment to an Active Test Sequence Step ..	<b>3-12</b>
Temporal Operators .....	<b>3-13</b>
Transition Operators .....	<b>3-15</b>
Use Messages in Test Sequences .....	<b>3-16</b>
<b>Generate Function-Based Test Signals</b> .....	<b>3-22</b>
Output Functions .....	<b>3-23</b>
<b>Assess Simulation Using Logical Statements</b> .....	<b>3-26</b>
verify .....	<b>3-26</b>
assert .....	<b>3-28</b>
Assessment Statements .....	<b>3-29</b>
Logical Operators .....	<b>3-30</b>
Relational Operators .....	<b>3-30</b>
<b>Syntax for Test Sequences and Assessments</b> .....	<b>3-32</b>
Assessment Statements .....	<b>3-29</b>
Temporal Operators .....	<b>3-13</b>
Transition Operators .....	<b>3-15</b>
Output Functions .....	<b>3-23</b>
Logical Operators .....	<b>3-30</b>
Relational Operators .....	<b>3-30</b>
<b>Debug a Test Sequence</b> .....	<b>3-40</b>
View Test Step Execution During Simulation .....	<b>3-40</b>
Set Breakpoints to Enable Debugging .....	<b>3-40</b>
View Data Values During Simulation .....	<b>3-41</b>
Step Through Simulation .....	<b>3-42</b>
<b>Test a Model Component Using Signal Functions</b> .....	<b>3-43</b>
Create a Test Sequence .....	<b>3-43</b>
Simulate the Test Harness .....	<b>3-45</b>
<b>Test Downshift Points of a Transmission Controller</b> .....	<b>3-47</b>
<b>Reuse Test Assessments</b> .....	<b>3-53</b>
Reuse Test Assessments Using a Library .....	<b>3-53</b>

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

4

<b>SIL Verification for a Subsystem</b> .....	4-2
Create a SIL Verification Harness for a Controller .....	4-3
Configure and Simulate a SIL Verification Harness .....	4-5
Compare the SIL Block and Model Controller Outputs .....	4-5

## Simulink Test Manager Introduction

5

<b>Introduction to the Test Manager</b> .....	5-2
Test Manager Description .....	5-2
Test Creation and Hierarchy .....	5-2
Test Results .....	5-3
Share Results .....	5-3

## Test Manager Test Cases

6

<b>Manage Test File Dependencies</b> .....	6-2
Package a Test File Using Simulink Projects .....	6-2
Find Test File Dependencies and Impact .....	6-4
Share a Test File with Dependencies .....	6-8
<b>Test Model Output Against a Baseline</b> .....	6-9
Create the Test Case .....	6-9
Run the Test Case and View Results .....	6-10
<b>Test a Simulation for Run-Time Errors</b> .....	6-13
Configure the Model .....	6-13
Create the Test Case .....	6-14
Run the Test Case .....	6-14
View Test Results .....	6-15

<b>Generate Test Cases from Model Components</b> .....	<b>6-16</b>
Generate the Test Cases .....	<b>6-16</b>
Synchronize Test Cases .....	<b>6-18</b>
Generate Test for a Subsystem .....	<b>6-20</b>
<b>Use External Inputs in Test Cases</b> .....	<b>6-24</b>
Use MAT-File for Inputs .....	<b>6-24</b>
Use Microsoft Excel File for Inputs .....	<b>6-24</b>
<b>Automate Tests Programmatically</b> .....	<b>6-27</b>
List of Functions and Classes .....	<b>6-27</b>
Create and Run a Test Case .....	<b>6-28</b>
<b>Run Multiple Combinations of Tests Using Iterations</b> . . . .	<b>6-30</b>
Create Table Iterations .....	<b>6-30</b>
Create Scripted Iterations .....	<b>6-33</b>
Sweep Through a Set of Parameters .....	<b>6-36</b>
<b>Collect Coverage in Tests</b> .....	<b>6-38</b>
<b>Run Tests Using Parallel Execution</b> .....	<b>6-44</b>
Use Parallel Execution .....	<b>6-44</b>
When Will Tests Benefit from Using Parallel Execution? . . .	<b>6-45</b>
<b>How Tolerances Are Applied to Test Criteria</b> .....	<b>6-46</b>
Modify Criteria Tolerances .....	<b>6-46</b>
<b>Test Manager Limitations</b> .....	<b>6-47</b>
Simulation Mode .....	<b>6-47</b>
Callback Scripts .....	<b>6-47</b>
Protected Models .....	<b>6-47</b>
Parameter Overrides .....	<b>6-48</b>
Breakpoints .....	<b>6-48</b>
Highlight in Model .....	<b>6-48</b>
<b>Test Sections</b> .....	<b>6-49</b>
Description .....	<b>6-50</b>
Requirements .....	<b>6-50</b>
System Under Test .....	<b>6-51</b>
Parameter Overrides .....	<b>6-52</b>
Callbacks .....	<b>6-52</b>
Inputs .....	<b>6-53</b>
Outputs .....	<b>6-54</b>

Configuration Settings . . . . .	6-54
Simulation 1 and Simulation 2 . . . . .	6-54
Equivalence Criteria . . . . .	6-54
Baseline Criteria . . . . .	6-55
Iterations . . . . .	6-55
Coverage Settings . . . . .	6-56

**Test Models Using Inputs Generated by Simulink Design**

<b>Verifier</b> . . . . .	6-57
Overall Workflow . . . . .	6-57
Test Case Generation Example . . . . .	6-57

**Test Manager Results and Reports**

**7**

<b>View Test Case Results</b> . . . . .	7-2
View Results Summary . . . . .	7-2
Visualize Test Case Simulation Output and Criteria . . . . .	7-4
<b>Export Test Results and Generate Reports</b> . . . . .	7-9
Export Results . . . . .	7-9
Create a Test Results Report . . . . .	7-10
Generate Report Using Microsoft Word Template . . . . .	7-10
<b>Customize Generated Reports</b> . . . . .	7-13
Inherit the Report Class . . . . .	7-13
Method Hierarchy . . . . .	7-13
Modify the Class . . . . .	7-15
Generate a Report Using the Custom Class . . . . .	7-17
<b>Results Sections</b> . . . . .	7-19
Summary . . . . .	7-20
Test Requirements . . . . .	7-20
Iteration Settings . . . . .	7-21
Errors . . . . .	7-21
Logs . . . . .	7-21
Description . . . . .	7-21
Parameter Overrides . . . . .	7-21
Coverage Results . . . . .	7-21



<b>Test Models in Real Time</b> .....	8-2
Overall Workflow .....	8-2
Real-Time Testing Considerations .....	8-3
Complete Basic Model Testing .....	8-3
Set up the Target Computer .....	8-3
Configure the Model or Test Harness .....	8-4
Add Test Cases for Real-Time Testing .....	8-6
Assess Real-Time Execution Using <code>verify</code> Statements ...	8-11



# Test Strategies

---

- “Functional Testing in Verification” on page 1-2
- “Link Tests to Requirements” on page 1-3

## Functional Testing in Verification

Model verification seeks to demonstrate that the “design is right,” that is, that the model meets the design requirements and conforms to standards. Model verification activities include property proving, model coverage measurement, requirements tracing, and functional testing.

Functional testing can be used across the model development cycle, and across levels of model complexity. An effective approach is to start with lower-level functional units and work up the model hierarchy to the system level. In functional testing, you simulate the model with one or more test cases and compare the result to expectations. Each test case includes inputs to the component under test, expected outputs, and test assessments. Rigorous functional testing maps each test case to a model requirement. Building up suites of test cases increases the range of requirements for which the model can be shown to behave as expected.

Functional testing can be used to:

- Test the model as it is being developed.
- Debug the model after completion.
- Check that the model does not regress.

Common methods of generating test inputs include logging signals from your model, writing test vectors based on requirements, or generating test cases using Simulink<sup>®</sup> Design Verifier<sup>™</sup>. You can define expected outputs using timeseries data and/or model assessments such as assertions. The goal is to provide a conclusive pass or fail result for your test.

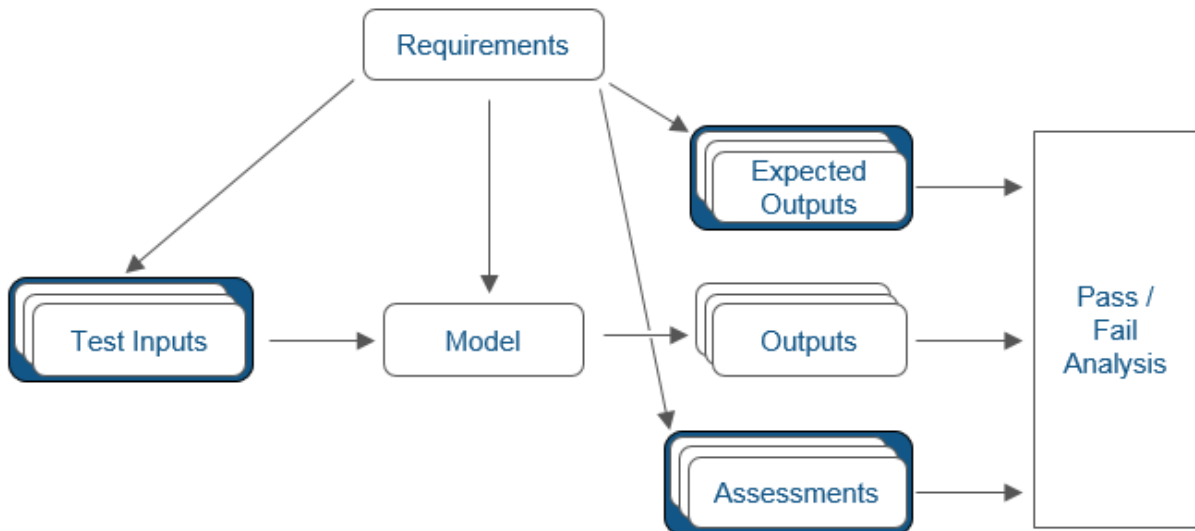
## Link Tests to Requirements

### In this section...

“Requirements Traceability Considerations” on page 1-3

“Establish Requirements Traceability for Testing” on page 1-4

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:

- Requirements linking is not supported for external 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 re-establishing 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” and “Requirements Traceability” in the Simulink Verification and Validation documentation.


### 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

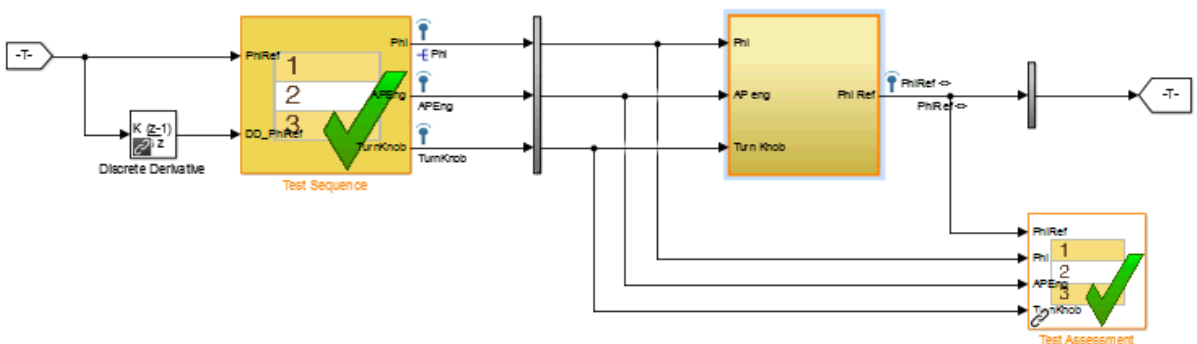
This example demonstrates adding requirements links to a test harness and test sequence. The model is a component of an autopilot roll control system. This example requires Simulink Test and Simulink Verification and Validation.

- 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

Requirements Document Index

New Up Down Delete Copy

DD Link

Description: DD Link

Document type: Text file Use current

Document: RollAutopilotRequirements.txt Browse...

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 3-53
- “Requirements-Based Testing for Model Development”

# Test Harness

---

- “Test Harness and Model Relationship” on page 2-2
- “Considerations and Limitations” on page 2-6
- “Select Test Harness Properties” on page 2-8
- “Test Harness Parameters and Signals” on page 2-12
- “Refine, Test, and Debug a Subsystem” on page 2-14
- “Manage Test Harnesses” on page 2-23
- “Synchronize Changes Between Test Harness and Model” on page 2-32
- “Test Library Blocks” on page 2-37

## Test Harness and Model Relationship

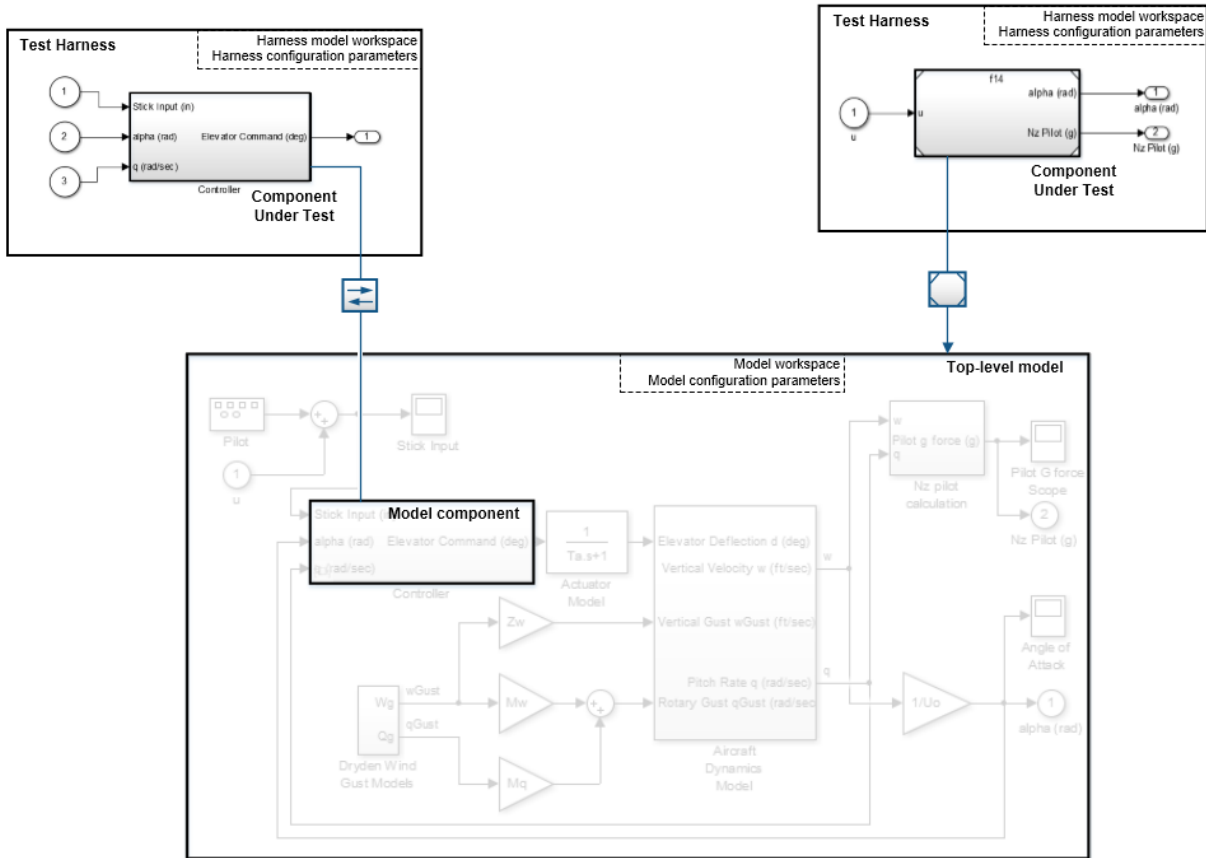
In this section...
“Test Harness Description” on page 2-2
“Harness — Model Relationship for a Model Component” on page 2-3
“Harness — Model Relationship for a Top-Level Model” on page 2-4
“Resolving Parameters” on page 2-5

### 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.

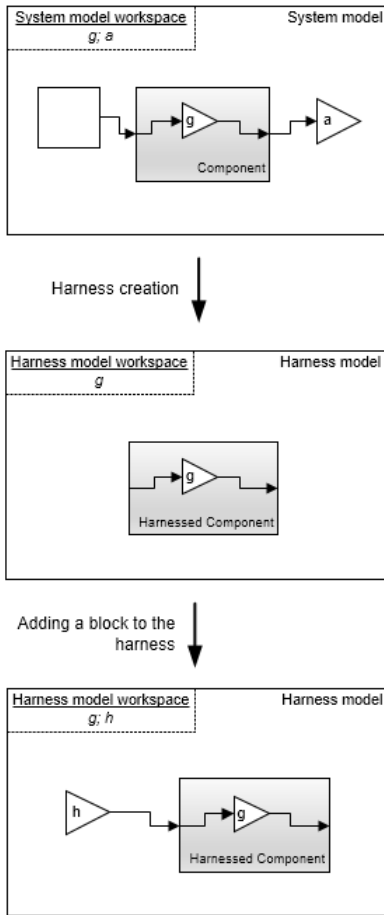


## Harness – Model Relationship for a Model Component

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

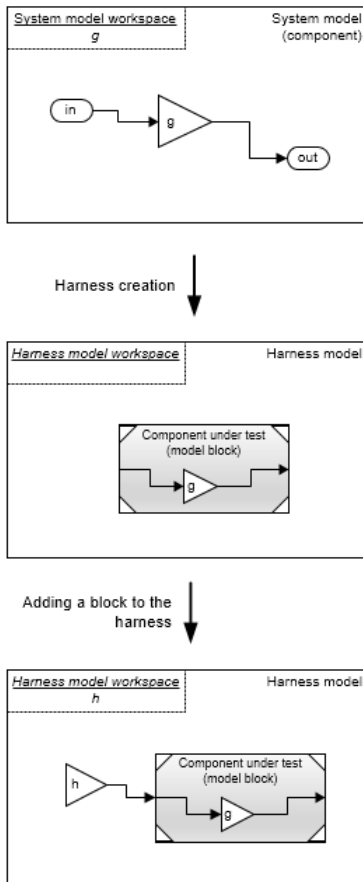
This example shows a test harness for a component that contains a Gain block. The harness model workspace contains a copy of the parameter  $g$  because  $g$  defines a part of the component.

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”

## 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” 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<sup>®</sup> 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
  - Arrays of buses
- 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” on page 2-8
“Considerations for Selecting Test Harness Properties” on page 2-8
“Save Test Harnesses Externally” on page 2-9
“Choosing Sources and Sinks” on page 2-9
“Use Separate Assessment Block” on page 2-9
“Initial Harness Configuration” on page 2-9
“Verification Modes” on page 2-11
“Change Harness Properties” on page 2-11

### 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.

## 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 2-23.

## 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 3-53.

## Initial Harness Configuration

You can select a preconfigured set of test harness properties for common tasks.

- **Prototyping:** Choose this configuration if your model is early in development. You can edit the component under test in the test harness, and control when the harness

is rebuilt from the main model. You can use this configuration if your main model does not compile.

- **Refinement/Debugging:** Choose this configuration if you want the test harness to include the configuration set, conversion subsystems, and model parameters for the component under test. This configuration can be useful for a nearly complete model, when you expect limited changes to the design.
- **Verification:** Choose this configuration if you require tight synchronization between the main model and the test harness, which is common for model verification. The test harness prevents you from editing the component under test, and the test harness rebuilds every time you open it. In addition to a normal subsystem, you can choose a SIL or PIL block as the component under test (requires Embedded Coder<sup>®</sup>). See “Verification Modes” on page 2-11.

You can also select a custom combination of harness properties. When you select **Custom**, these options become available:

Property	Description	Additional Information
<b>Create without compiling the model</b>	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.
<b>Rebuild harness on open</b>	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 2-32.
<b>Update Configuration Parameters and Model Workspace data on rebuild</b>	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 2-32.
<b>Enable component editing in harness model</b>	When you select this property, you can edit the component under test in the test harness.	

## 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 2-32

## Test Harness Parameters and Signals

**In this section...**

“Test Harness Generation Without Compilation” on page 2-12

“Signal Conversion Subsystem” on page 2-12

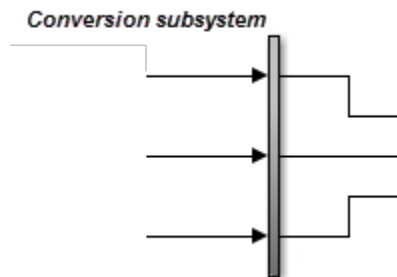
### Test Harness Generation Without Compilation

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 2-32.

### Signal Conversion Subsystem



A signal conversion subsystem contains signal specification blocks to check signal properties to and from the component under test, such as:

- Data type
- Sample time
- Bus properties
- Dimension
- Complexity

Like the main model, a test harness does not compile if the signal types do not match the signal specification. If you get a compile error related to the signal conversion subsystem, check the signal properties and modify the test harness design if necessary. For example:

- You can add conversion blocks to your test harness outside the conversion subsystem.
- You can 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. Changes made to the conversion subsystems are lost.

---

## Refine, Test, and Debug a Subsystem

In this section...
“Model and Requirements” on page 2-14
“Create a Harness for the Controller” on page 2-16
“Inspect and Refine the Controller” on page 2-18
“Add a Test Case and Test the Controller” on page 2-19
“Debug the Controller” on page 2-20

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'))
```

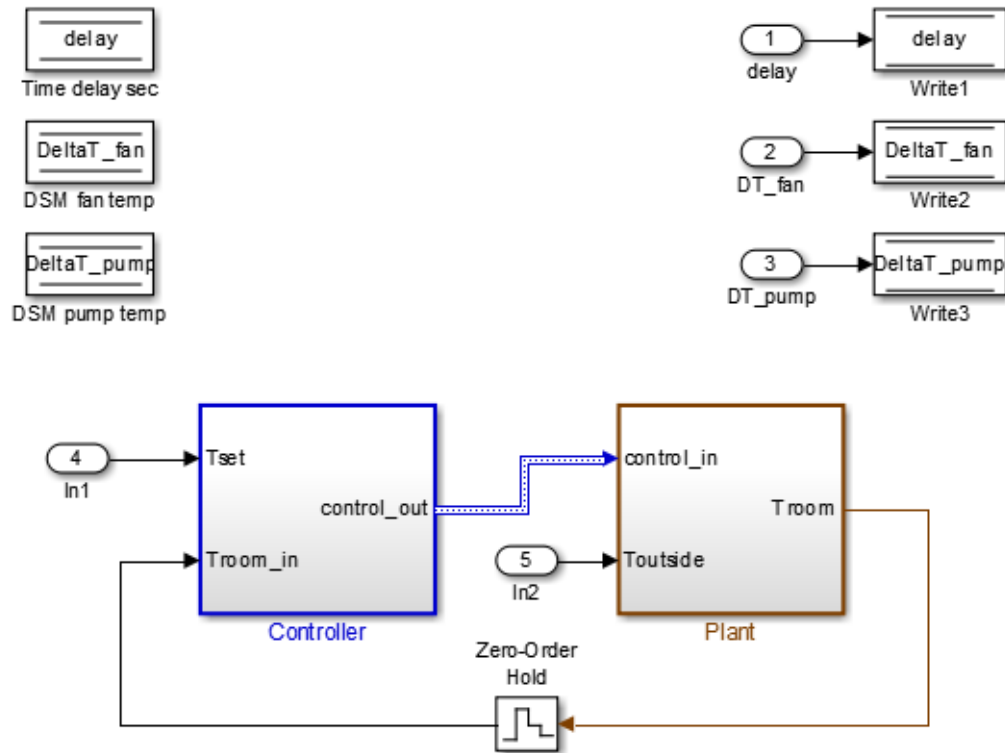
- 2 Copy this model file and supporting files to a writable location on the MATLAB path:

```
sltestHeatpumpExample.slx  
sltestHeatpumpBusPostLoadFcn.mat  
PumpDirection.m
```

- 3 Open the model.

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





Copyright 1990-2014 The MathWorks, Inc.

In the example model:

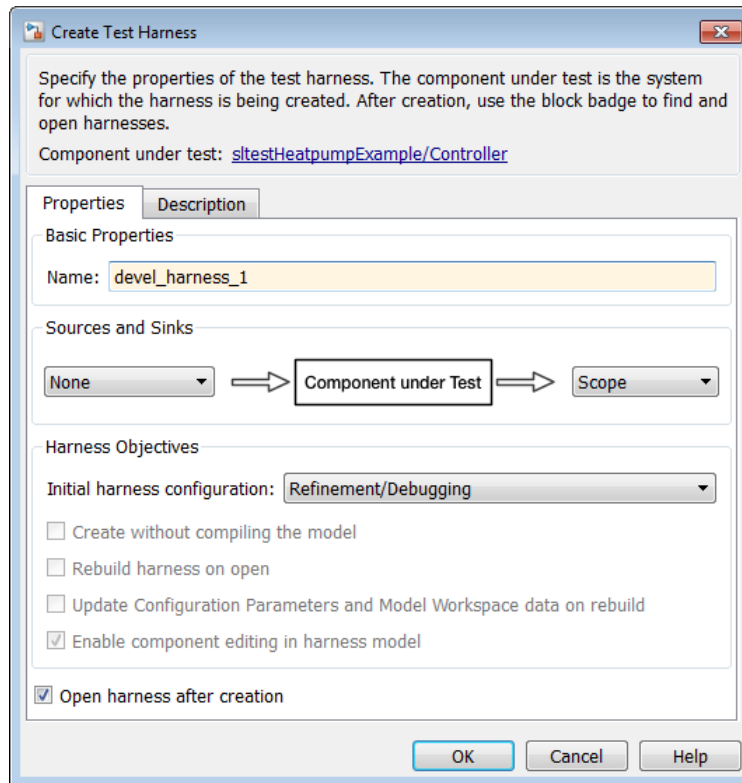
- The controller accepts the room temperature and the set temperature inputs.
- The controller output is a bus with signals controlling the fan, heat pump, and the direction of the heat pump (heat or cool).
- The plant accepts the control bus. The heat pump and the fan signals are Boolean, and the heat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

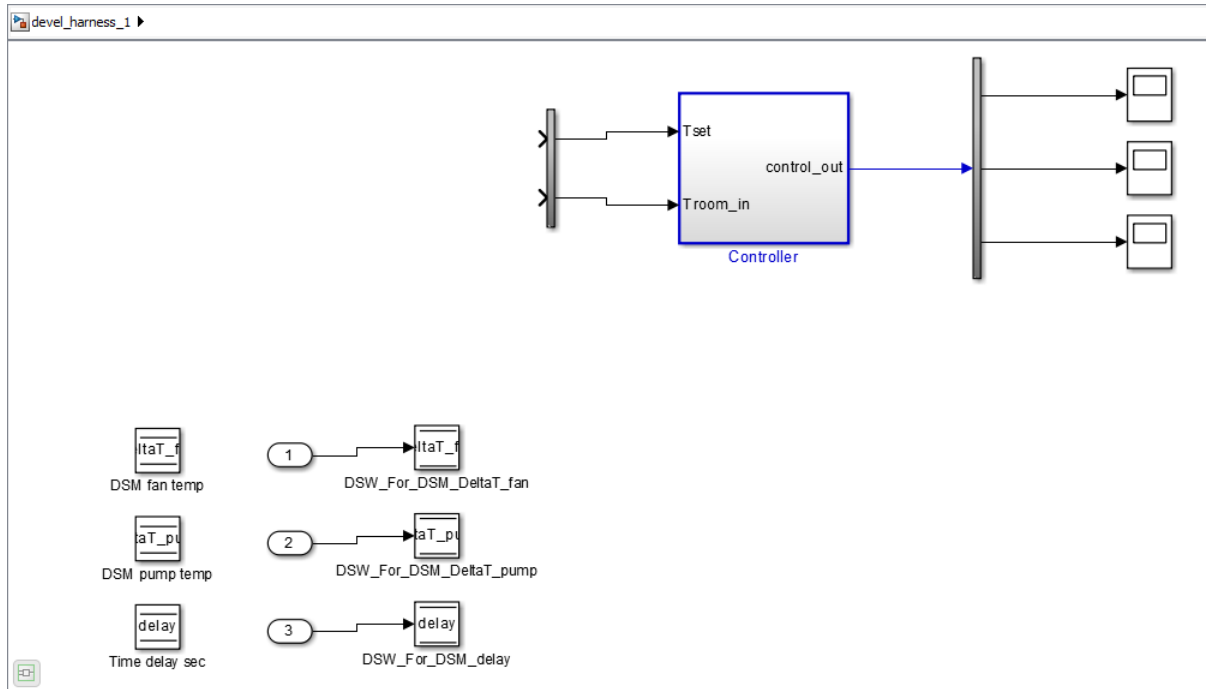
Temperature condition	System state	Fan command	Pump command	Pump direction
$ T_{room} - T_{set}  < \Delta T_{fan}$	idle	0	0	0
$\Delta T_{fan} \leq  T_{room} - T_{set}  < \Delta T_{pump}$	fan only	1	0	0
$ T_{room} - T_{set}  < \Delta T_{pump}$ and $T_{set} < T_{room}$	cooling	1	1	-1
$ T_{room} - T_{set}  < \Delta T_{pump}$ and $T_{set} > T_{room}$	heating	1	1	1

### Create a Harness for the Controller

- 1 Right-click the Controller subsystem and select **Test Harness > Create Test Harness (Controller)**.
- 2 Set the harness properties:
  - **Name:** devel\_harness\_1
  - **Sources and Sinks:** None and **Scope**
  - **Add separate assessment block:** Cleared
  - **Initial harness configuration:** Refinement/Debugging
  - Select **Open harness after creation**.

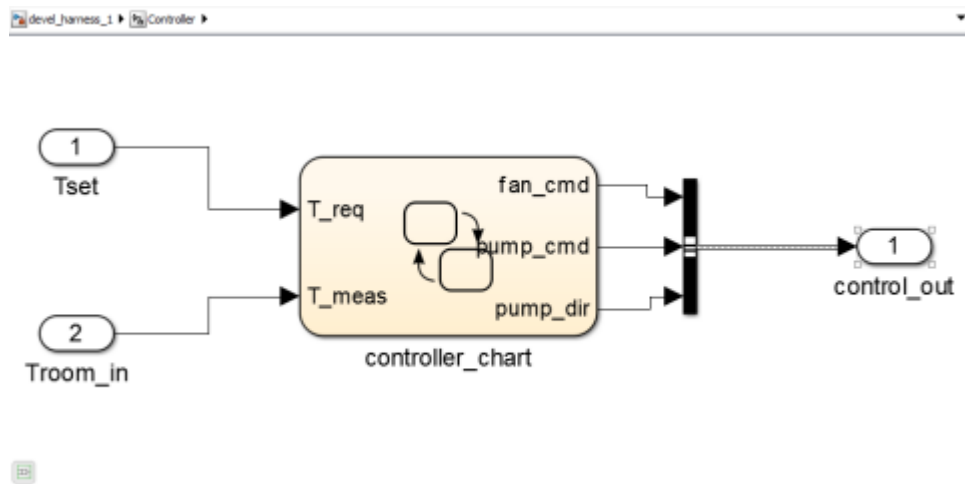


- 3 Click **OK** to create the test harness.



### Inspect and Refine the Controller

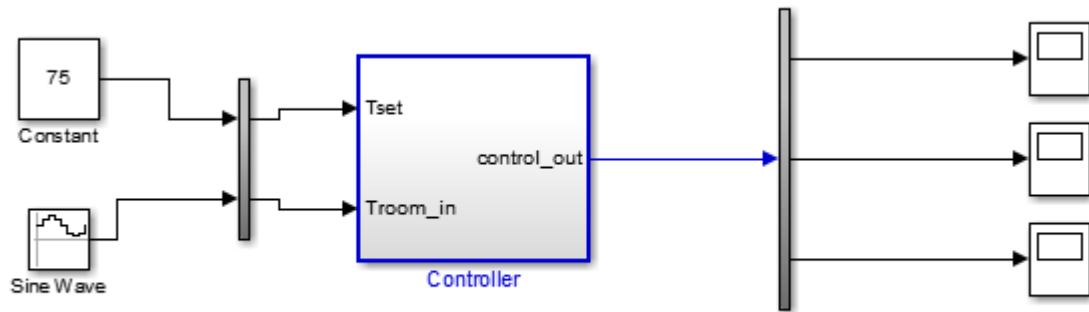
- 1 Double-click Controller to open the subsystem.
- 2 Notice that the chart is disconnected from its ports. Fix this issue by connecting the chart as shown.



- 3 In the harness, click the save button in the toolbar to save the 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 case 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**.

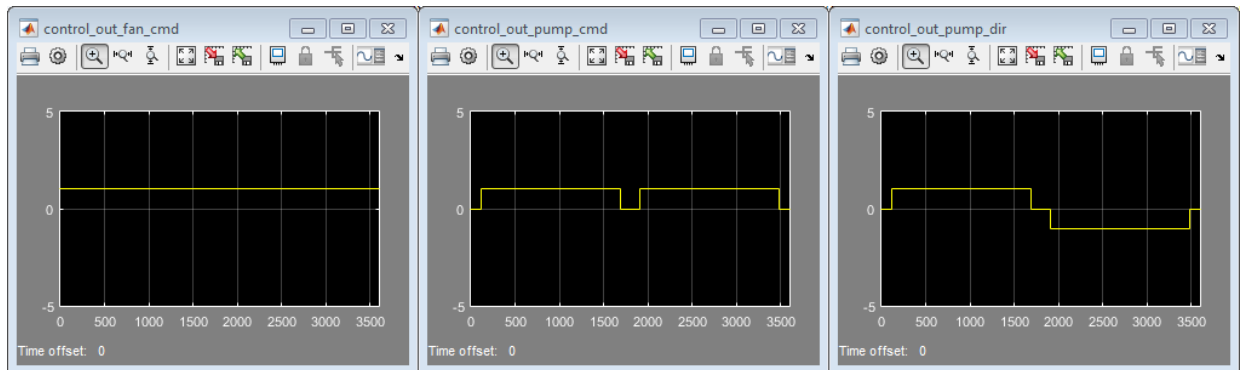


- 5 In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select **Input** and enter `u`. `u` is an existing structure in the MATLAB base workspace.
- 6 In the **Solver** pane, set **Stop time** to 3600.
- 7 Open the three scopes in the harness model.
- 8 Simulate the harness.

### Debug the Controller

- 1 Observe the controller output. `fan_cmd` is 1 during the IDLE condition where  $|T_{room} - T_{set}| < \Delta T_{fan}$ .

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



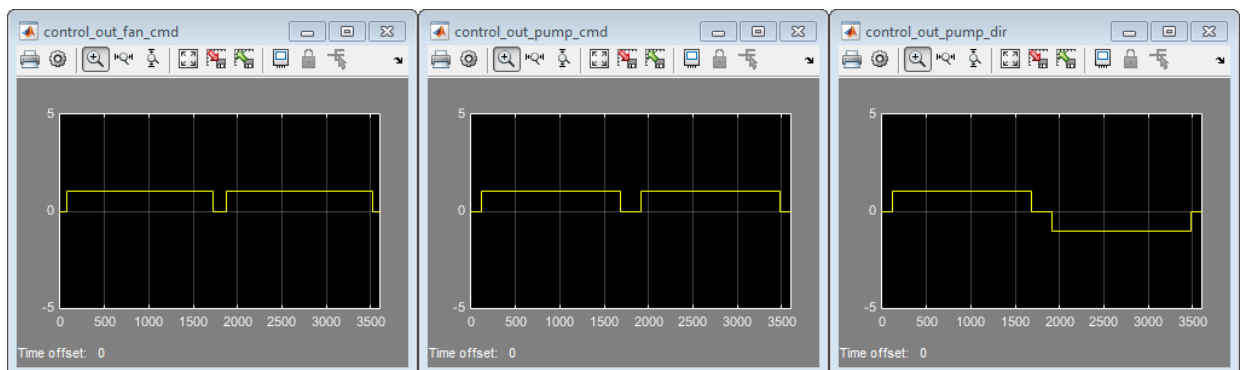
- 2 In the harness model, open the Controller subsystem.
- 3 Open `controller_chart`.
- 4 In the IDLE state, `fan_cmd` is set to return 1. Change `fan_cmd` to return 0. IDLE is now:

```

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 3-43
- “Test Downshift Points of a Transmission Controller” on page 3-47



## Manage Test Harnesses

### In this section...

“Internal and External Test Harnesses” on page 2-23

“Manage External Test Harnesses” on page 2-23

“Convert Between Internal and External Test Harnesses” on page 2-24

“Preview and Open a Test Harness” on page 2-26

“Find Test Cases Associated with a Test Harness” on page 2-27

“Export Test Harnesses to Separate Models” on page 2-27

“Clone and Export a Test Harness to a Separate Model” on page 2-28

“Delete Test Harnesses in a Model” on page 2-30

### 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:

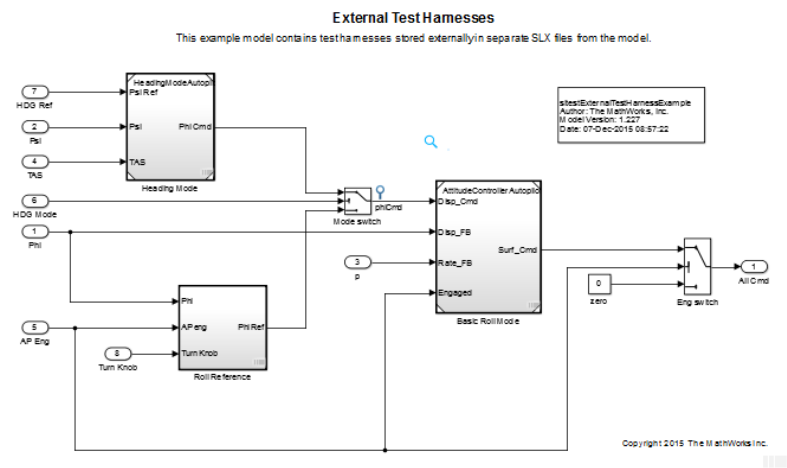
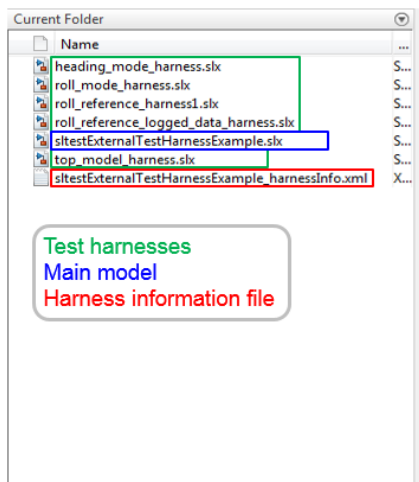
- Develop your model using internal test harnesses so that you can more easily share the model for review. When you complete your design and place the model under change control, convert to external harnesses.
- Use the change-controlled model as the starting point for a new design. Test the existing 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):

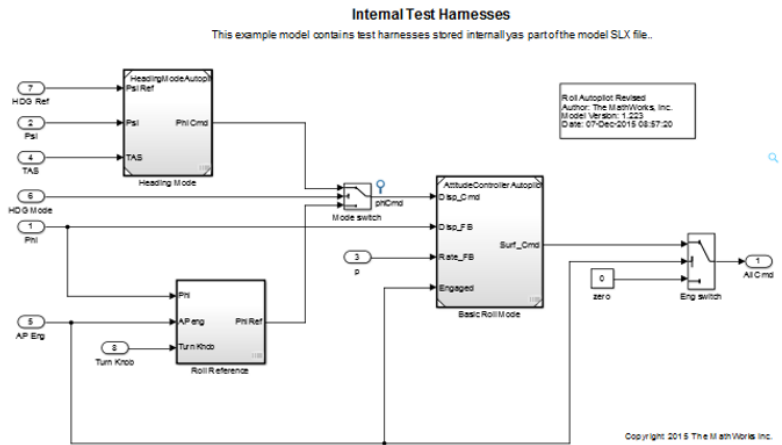
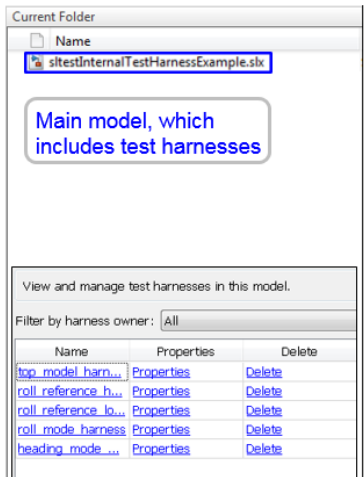
- 1 Navigate to the top of the main model.
- 2 Select **Analysis > Test Harness > Convert To External (Internal) Harnesses**.
- 3 A dialog box provides information on the conversion procedure and the affected test harnesses. Click **Yes** to continue.

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`.

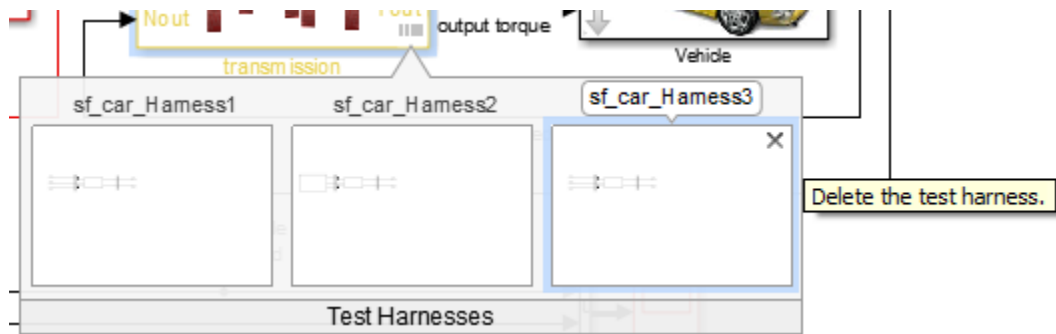


Inversely, conversion to internal test harnesses moves the test harness SLX files and the harnessInfo.xml file.

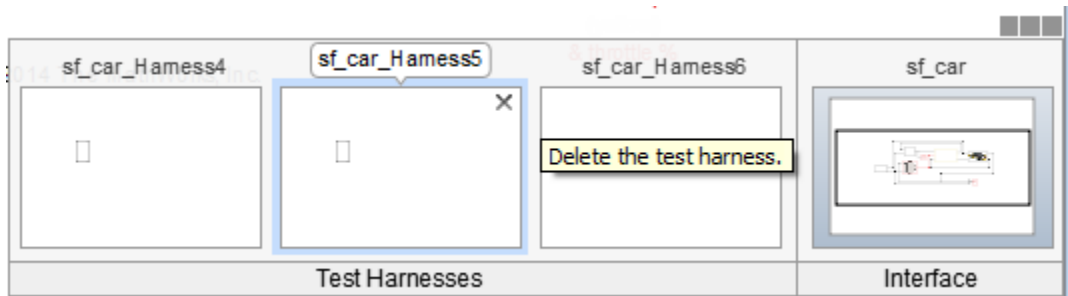


## Preview and Open a Test Harness


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.

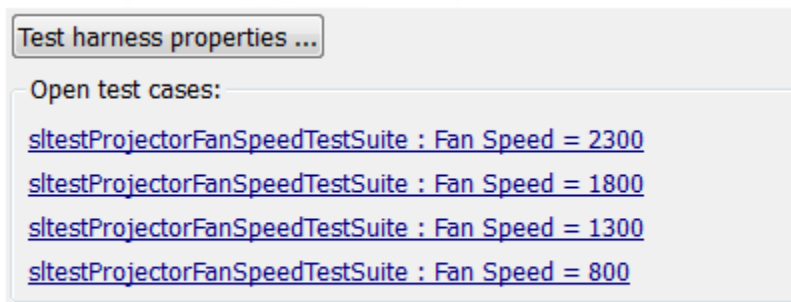


When a model block diagram has a test harness, click the pullout icon in the model canvas to preview the test harnesses. To open the harness, click a thumbnail.



## 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.



## 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.

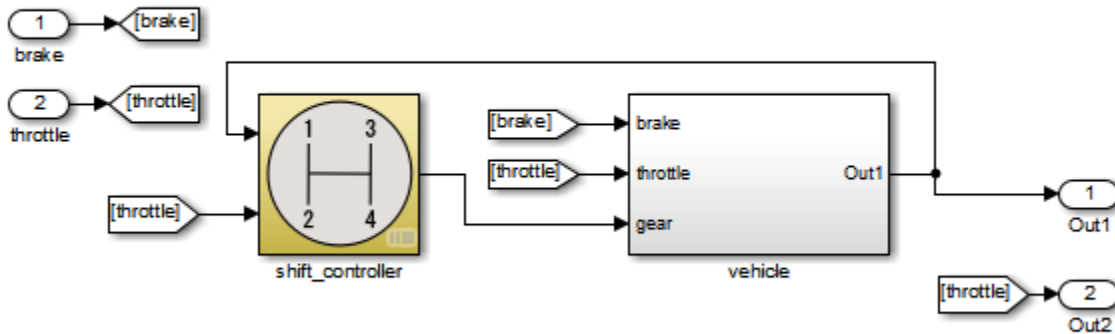
### Open the Model and Save a Local Copy

```
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 2015 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 =
```

```

    model: 'sltestTestSequenceExample'
    name: 'controller_harness'
  description: ''
    type: 'Testing'
  ownerHandle: 12.0013
  ownerFullPath: 'sltestTestSequenceExample/shift_controller'
  ownerType: 'Simulink.SubSystem'
    isOpen: 0
  canBeOpened: 1
    lockMode: 0
  verificationMode: 0
  saveIndependently: 0
    rebuildOnOpen: 0
  rebuildModelData: 0
```

```
graphical: 0
  origSrc: 'Test Sequence'
  origSink: 'Test Assessment'
```

### 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')
open_system('ControllerTestModel')

clear('Model');
bdclose all;
```

### Delete Test Harnesses in a Model

You can delete a harness manually, using harness options icon  in the harness thumbnail. You can also delete harnesses programmatically, which can reduce effort when your model has harnesses at different hierarchy levels. This example demonstrates creating four test harnesses for a model and deleting them.

- 1 Open the model.

```
open_system('sf_car');
```

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

```
sltest.harness.create('sf_car/transmission');
sltest.harness.create('sf_car/transmission');
```



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

- 3** Find the harnesses in the `sf_car` model.

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

```
test_harness_list =
```

```
1x4 struct array with fields:
```

```
    model
    name
    description
    type
    ownerHandle
    ownerFullPath
    ownerType
    isActive
    canBeActivated
    lockMode
    verificationMode
    saveIndependently
    rebuildOnOpen
    rebuildModelData
```

- 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
```

## See Also

### Functions

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

## Synchronize Changes Between Test Harness and Model

### In this section...

“Maintain SIL or PIL Block Fidelity” on page 2-32

“Synchronize Changes to the Component Under Test” on page 2-32

“Rebuild Test Harness” on page 2-33

“Update Parameters from Test Harness to Model” on page 2-33

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. These abilities provide an advantage over isolating a model component in a separate Simulink model.

### 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 SIL and PIL Simulations”.

### Synchronize Changes to the Component Under Test

The component in the harness or the main model updates to the latest design when you open or close a test harness:

- Design changes from model to harness — The component under test updates when you open the harness.
- Design changes from harness to model — The component in the model updates when you close the harness.

---

**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.

---

## 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'))
```

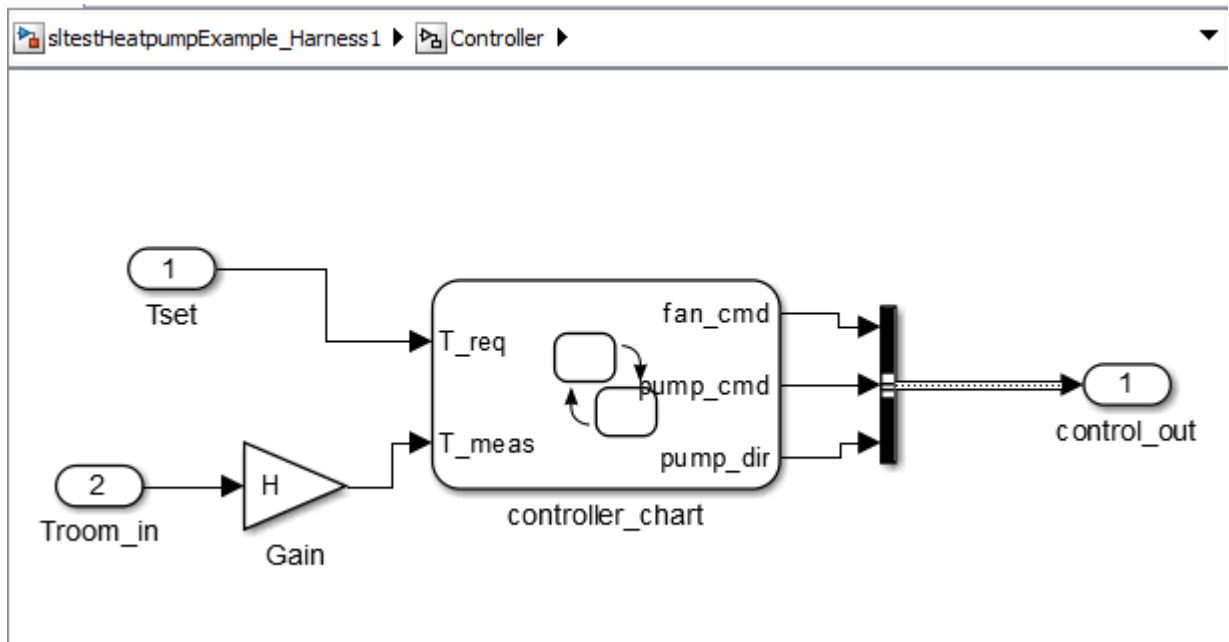
- 2 Copy this model file and supporting files to a writable location on the MATLAB path:

```
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:  [Show Details](#)

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...**

“Library Testing Workflow” on page 2-37

“Library and Linked Subsystem Test Harness” on page 2-38

“Edit Library Block from a Test Harness” on page 2-39

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.

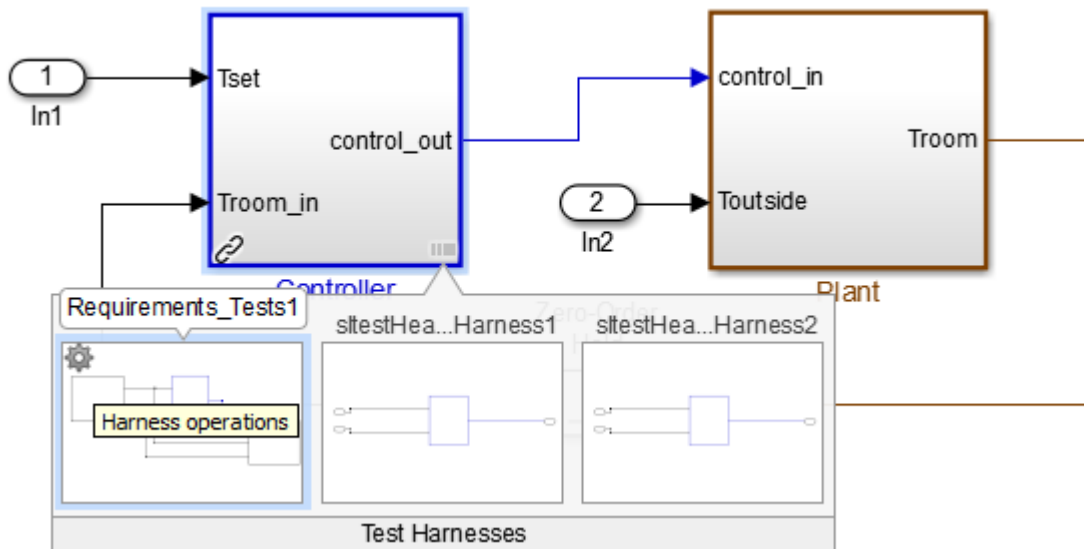
- Libraries do not compile, so a test harness for a library subsystem does not contain compiled attributes.
- A test harness for a library subsystem does not generate conversion subsystems for the block inputs and outputs.
- A library subsystem test harness does not use push or rebuild operations, because 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 2-23.

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

- “Testing a Library and a Linked Block”



# Test Sequences and Assessments

---

- “Introduction to Test Sequences” on page 3-2
- “Organize Test Sequences” on page 3-8
- “Test Sequence Action and Transition Operations” on page 3-11
- “Generate Function-Based Test Signals” on page 3-22
- “Assess Simulation Using Logical Statements” on page 3-26
- “Syntax for Test Sequences and Assessments” on page 3-32
- “Debug a Test Sequence” on page 3-40
- “Test a Model Component Using Signal Functions” on page 3-43
- “Test Downshift Points of a Transmission Controller” on page 3-47
- “Reuse Test Assessments” on page 3-53

# Introduction to Test Sequences

### In this section...

“Structure of a Test Sequence” on page 3-2

“Test Sequence Hierarchy” on page 3-2

“Step Transitions” on page 3-2

“Create a Basic Test Sequence” on page 3-3

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.

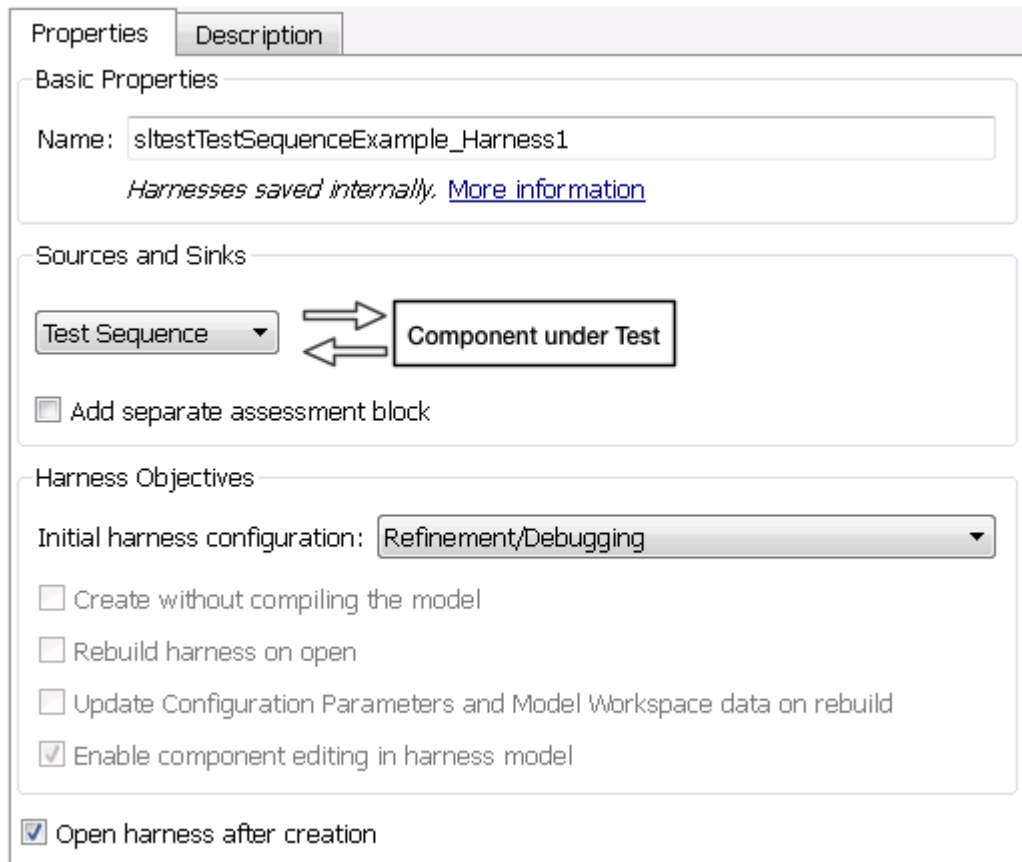
## Create a Basic Test Sequence

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.



- 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)
```

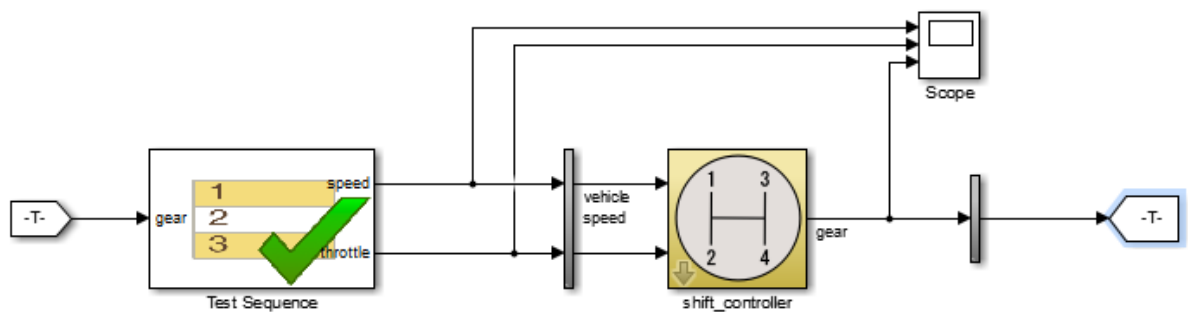
```
Else
```

The fourth step **Else** takes no action. **Else** handles conditions that make no other **when** statement valid.

### 3 Test Sequences and Assessments

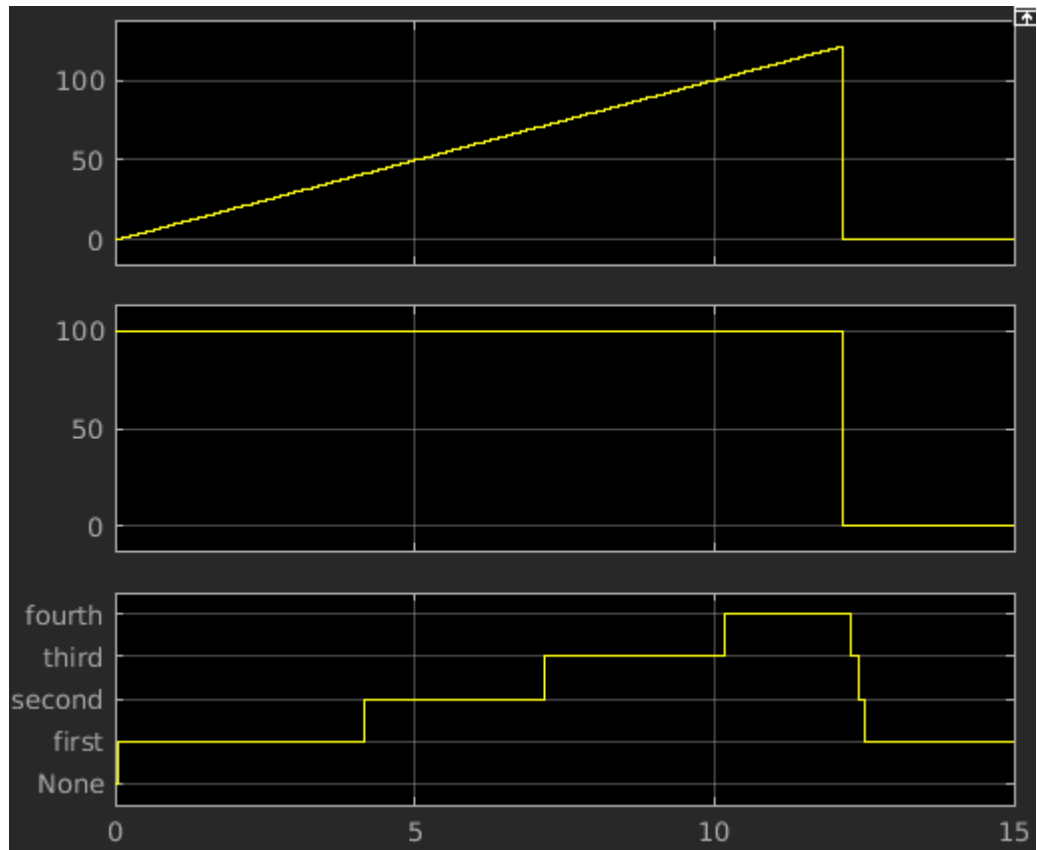
Symbols	Step	Transition	Next Step
<b>Input</b> 1. gear	<b>Accelerate</b> speed = 10*ramp(et); throttle = 100;	1. duration(gear == 4) >= Limit	Stop
<b>Output</b> 1. speed 2. throttle	Check1st when gear == 1 verify(speed < 45)		
<b>Local</b>	Check2nd when gear == 2 verify(speed < 75)		
<b>Constant</b> Limit	Check3rd when gear == 3 verify(speed < 105)		
<b>Parameter</b>	Else		
<b>Data Store Memory</b>	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**

“Syntax for Test Sequences and Assessments” on page 3-32 | Test Sequence

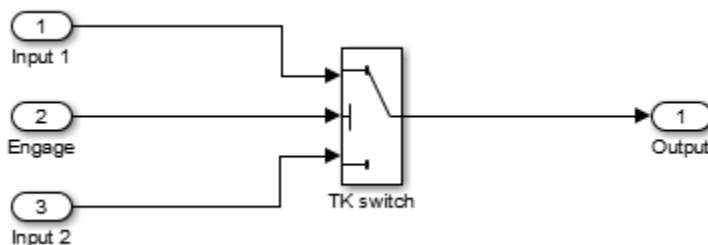
## Organize Test Sequences

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	Step	Transition	Next Step
<b>Input</b> SwitchOutput	InitializeTest Input1 = 0; Engage = 0; Input2 = 0; EndTest = 0;	1. Input1 == 0 &&... Input2 == 0 &&... Engage == 0	OffOn_Test
<b>Output</b> Engage Input1 Input2	OffOn_Test	1. EndTest == 1	OnOff_Test
<b>Local</b> EndTest	SetSignals Input1 = SignalLow; Input2 = SignalHigh; Engage = 0;	1. true	Engage_Low_High
<b>Constant</b> SignalHigh SignalLow	Engage_Low_High Engage = 1;	1. true	Assess_Low_High
<b>Parameter</b>	Assess_Low_High assert(SwitchOutput == Input1);	1. true	EndTest
<b>Data Store Memory</b>	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” on page 3-11

“Link a Test Assessment to an Active Test Sequence Step” on page 3-12

“Temporal Operators” on page 3-13

“Transition Operators” on page 3-15

“Use Messages in Test Sequences” on page 3-16

### 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

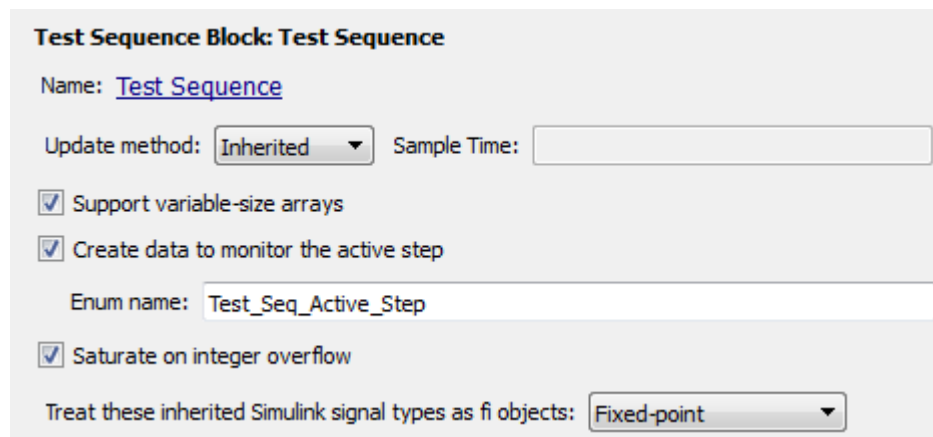
Data Symbols	Step	Transition	Next Step
<b>Input</b> Switch	<b>Initialize</b> SignalOut = 0;	1. true	Sine ▼
<b>Output</b> SignalOut	<b>Sine</b> SignalOut = sin(et*2*pi/10);	1. hasChanged(Switch)	Sine8 ▼
<b>Local</b>	<b>Sine8</b> SignalOut = sin(et*8*pi/10);	1. hasChangedFrom(Switch,1)	Sine16 ▼
<b>Constant</b>	<b>Sine16</b> SignalOut = sin(et*16*pi/10);	1. hasChangedTo(Switch,13.344)	Stop ▼
<b>Parameter</b>	<b>Stop</b> SignalOut = 0;		
<b>Data Store Memory</b>			

## 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:

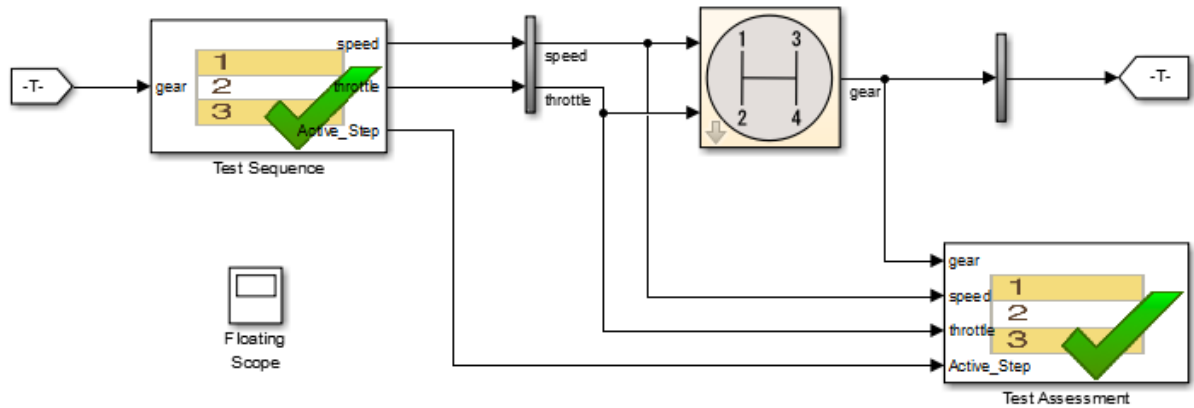
- 1 Open the Model Explorer by selecting **View > Model Explorer > Model Explorer**.
- 2 Select the **Test Sequence** block in the **Model Hierarchy**.
- 3 In the properties, select **Create data to monitor the active step**.

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



The screenshot shows the 'Test Sequence Block: Test Sequence' properties dialog. The 'Name' field is 'Test Sequence'. The 'Update method' is 'Inherited' and 'Sample Time' is empty. There are two checked checkboxes: 'Support variable-size arrays' and 'Create data to monitor the active step'. The 'Enum name' field contains 'Test\_Seq\_Active\_Step'. There is also a checked checkbox for 'Saturate on integer overflow' and a dropdown for 'Treat these inherited Simulink signal types as fi objects' set to '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  $\approx$  2.

Step	Transition	Next Step
<div style="border: 1px solid #ccc; padding: 5px;"> <p><b>AssertConditions</b></p> <pre> assert(speed &gt;= 0); assert(throttle &gt;= 0); assert(throttle &lt;= 100); assert(gear &gt; 0);                     </pre> </div>		
<div style="border: 1px solid #ccc; padding: 5px;"> <p><b>verify_4 when Active_Step == Test_Seq_Active_Step.down_4_3</b></p> <pre> verify(gear <math>\approx</math> 2);                     </pre> </div>		
<div style="border: 1px solid #ccc; padding: 5px;"> <p><b>Else</b></p> </div>		

## 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. Syntax in the table uses these arguments:

#### TimeUnits

The units of time

Value: sec | msec | usec

Examples:

msec

#### SignalCondition

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`

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



Operator	Syntax	Description	Example
duration	ElapsedTime = d	duration returns ElapsedTime in TimeUnits since SignalCondition 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

## Transition Operators

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

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. u cannot be an expression or other type of variable.	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. u cannot be an expression or other type of variable.	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. u cannot be an	Transition when h changes to 0:  hasChangedTo(h,0)

Operator	Syntax	Description	Example
		expression or other type of variable.	

## 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.

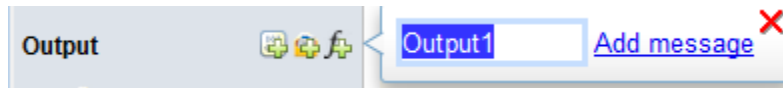


`receive(M)` determines whether a message is present in the input queue `M`, and removes the message from the queue. `receive(M)` returns `true` if a message is in the queue, and `false` if not. Once the message is received, you can access the message data using the dot notation, `M.data`, or forward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

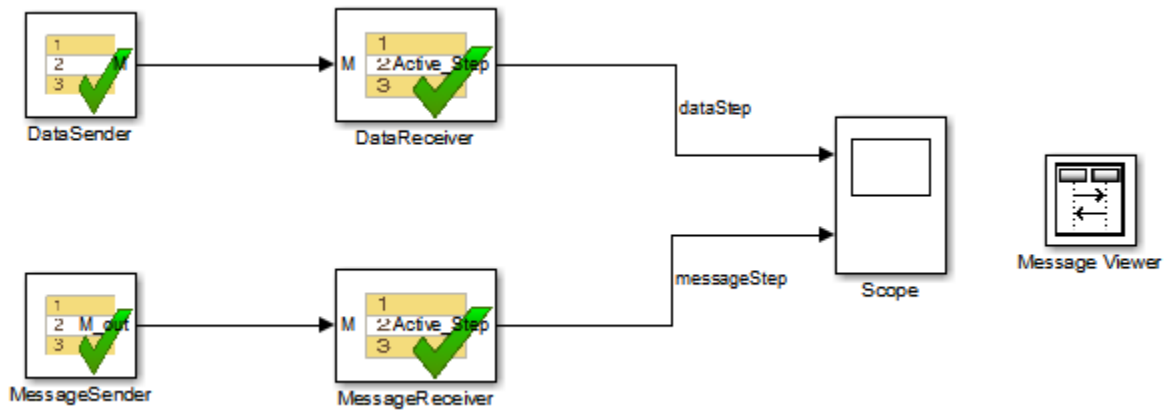
This example demonstrates message inputs and outputs, sending, and receiving a message. The model compares two pairs of test sequences. Each pair is comprised of a sending and receiving test sequence block. The first pair sends and receives data, and the second sends and receives a message.

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. <code>after(3,sec)</code>	S2 ▼	Waits
S2	1. <code>M == 3.5</code>	S3 ▼	
S3	1. <code>M == 3.5</code>	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 <code>M.data = 3.5;</code>	1. <code>true</code>	step_2 ▼	Assigns a value to the message's data
step_2 <code>send(M)</code>	1. <code>true</code>	step_3 ▼	Sends the message
step_3			

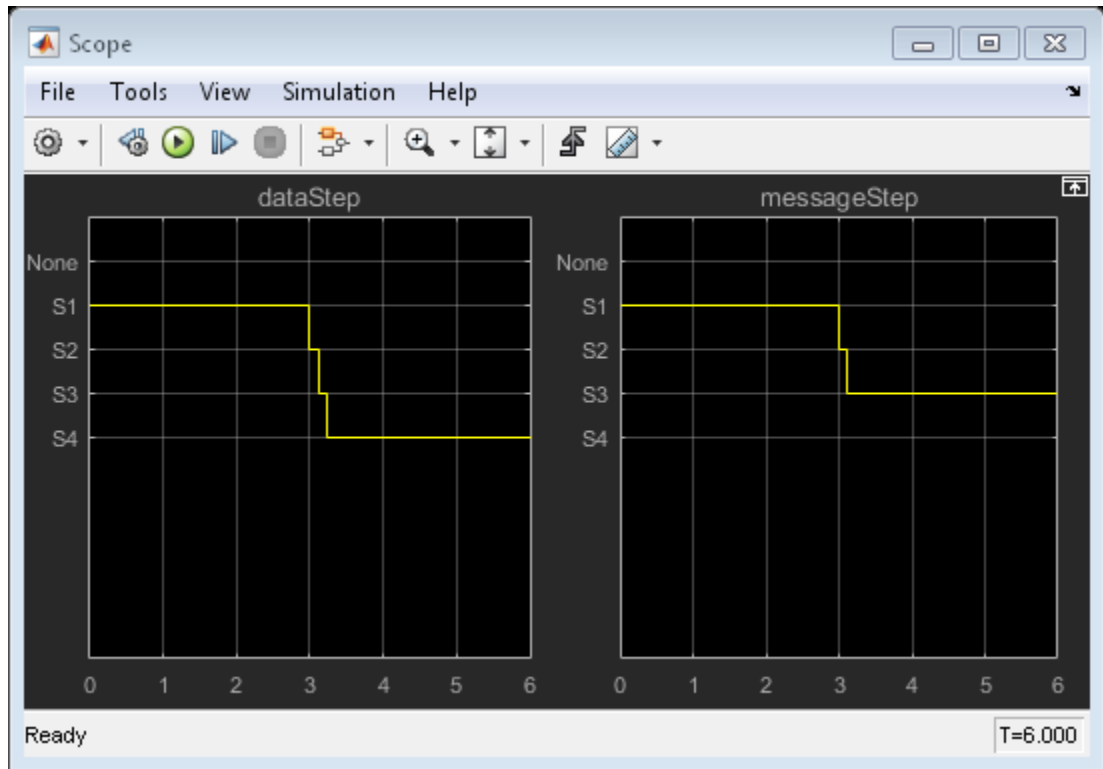
The MessageReceiver block waits 3 seconds, then transitions to step S2. Step S2's transition evaluates the queue `M` with `receive(M)`, removing the message from the queue. `receive(M)` returns `true` since the message is present. `M.data == 3.5` compares the message data to the expected value. The statement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

Step	Transition	Next Step	Description
S1	1. <code>after(3,sec)</code>	S2	▼ Waits.
S2	1. <code>receive(M) &amp;&amp; M.data == 3.5</code>	S3	▼ Transitions to S3 if a message is available in the queue and message data == 3.5.
S3	1. <code>receive(M)</code>	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 3-32 | Test Sequence

## Related Examples

- “Generate Function-Based Test Signals” on page 3-22
- “Assess Simulation Using Logical Statements” on page 3-26

## 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.

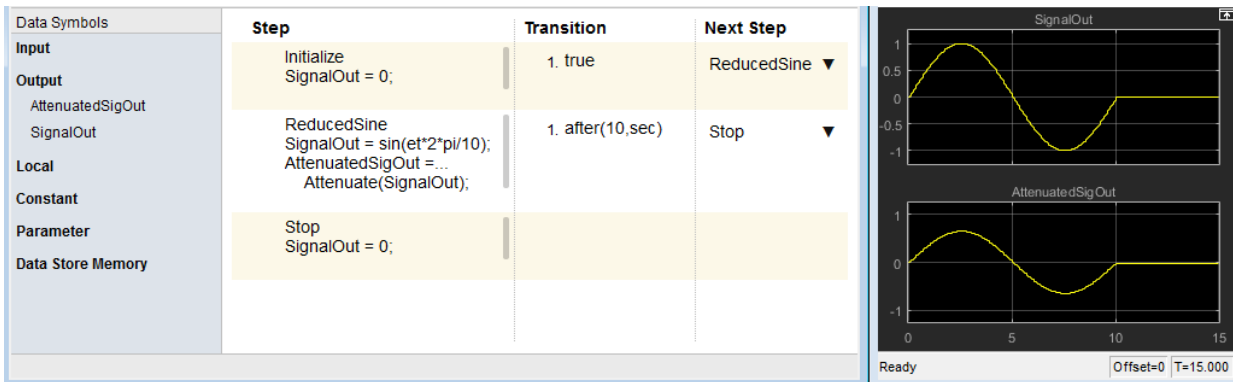
In this test sequence, the step **Sine** outputs a sine wave with a period of 10 seconds, specified by the argument  $et*2*pi/10$ . The step **Random** outputs a random number in the interval -0.5 to 0.5.



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
```





## Output Functions

You can use functions to generate test signals. 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.

Function	Syntax	Description	Example
square	<code>square(x)</code>	Represents a square wave output with a period of 1 and range -1 to 1.  Within the interval $0 \leq x < 1$ , <code>square(x)</code> returns the value 1 for $0 \leq x < 0.5$ and -1 for $0.5 \leq x < 1$ .	Output a square wave with a period of 10 sec:  <code>square(et/10)</code>
sawtooth	<code>sawtooth(x)</code>	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:  <code>sawtooth(et/10)</code>

Function	Syntax	Description	Example
		Within the interval $0 \leq x < 1$ , <code>sawtooth(x)</code> increases.	
<code>triangle</code>	<code>triangle(x)</code>	Represents a triangle wave output with a period of 1 and range -1 to 1.  Within the interval $0 \leq x < 0.5$ , <code>triangle(x)</code> increases.	Output a triangle wave with a period of 10 sec:  <code>triangle(et/10)</code>
<code>ramp</code>	<code>ramp(x)</code>	Represents a ramp signal of slope 1, returning the value of the ramp at time $x$ .  <code>ramp(et)</code> effectively returns the elapsed time of the test step.	Ramp one unit for every five seconds of test step elapsed time:  <code>ramp(et/5)</code>
<code>heaviside</code>	<code>heaviside(x)</code>	Represents a heaviside step signal, returning 0 for $x < 0$ and 1 for $x \geq 0$ .	Output a heaviside signal after 5 seconds:  <code>heaviside(et-5)</code>
<code>latch</code>	<code>latch(x)</code>	Returns the current value of $x$ and holds that value during the current test step.	Latch <code>b</code> to the value of torque:  <code>b = latch(torque)</code>
<code>sin</code>	<code>sin(x)</code>	Returns the sine of $x$ , where $x$ is in radians.	A sine wave with a period of 10 sec:  <code>sin(et*2*pi/10)</code>
<code>cos</code>	<code>cos(x)</code>	Returns the sine of $x$ , where $x$ is in radians.	A cosine wave with a period of 10 sec:  <code>cos(et*2*pi/10)</code>

Function	Syntax	Description	Example
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 1/10th the test step elapsed time:  $exp(et/10)$

## See Also

“Syntax for Test Sequences and Assessments” on page 3-32 | Test Sequence

## Related Examples

- “Test Sequence Action and Transition Operations” on page 3-11
- “Assess Simulation Using Logical Statements” on page 3-26

## Assess Simulation Using Logical Statements

In this section...
“verify” on page 3-26
“assert” on page 3-28
“Assessment Statements” on page 3-29
“Logical Operators” on page 3-30
“Relational Operators” on page 3-30

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 never fails, and at any time the statement passes, the overall result passes. Otherwise, the statement is never 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.

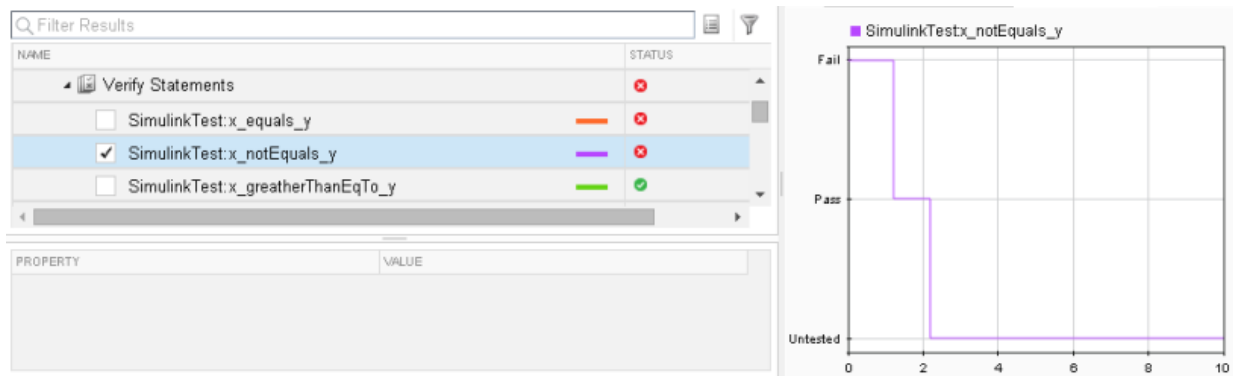
### Example

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
<div style="border: 1px solid #ccc; padding: 5px;"> <div style="display: flex; align-items: flex-start;"> <div style="width: 20px; border-right: 1px solid #ccc; margin-right: 5px;"></div> <div> <p>Comparison_example</p> <pre>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 &gt;= y, 'SimulinkTest:x_greaterThanEqTo_y','x and y values are %d, %d',x,y)</pre> </div> </div> </div>	1. testFlag == 1	End ▼
<div style="border: 1px solid #ccc; padding: 5px;"> <div style="display: flex; align-items: flex-start;"> <div style="width: 20px; border-right: 1px solid #ccc; margin-right: 5px;"></div> <div> <p>step_1_1</p> <pre>x = 2; y = 2;</pre> </div> </div> </div>	1. after(1,sec)	step_1_2 ▼
<div style="border: 1px solid #ccc; padding: 5px;"> <div style="display: flex; align-items: flex-start;"> <div style="width: 20px; border-right: 1px solid #ccc; margin-right: 5px;"></div> <div> <p>step_1_2</p> <pre>x = 5; y = 2;</pre> </div> </div> </div>	1. after(1,sec)	change ▼

The test manager displays the results.



## 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
<pre>InitializeCheck assert(h == 0 &amp;&amp; k == 0, 'Signals must initialize to 0');</pre>		
<pre>step_1 test_output = true;</pre>	<pre>1. after(1,sec)</pre>	<pre>step_2 ▼</pre>

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

## Assessment Statements

You can use assessment statements inside a **Test Sequence** or **Test Assessment** block to verify simulation, stop simulation, and return verification results. 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
```

Keyword	Statement Syntax	Description	Example
verify	<pre>verify(expression) verify(expression, e verify(expression, i</pre>	Assesses a logical expression. Optional arguments label results in the test manager and diagnostic viewer.	<pre>verify(x &gt; y, 'SimulinkTest:gre 'x and y values are %d, %d',x,y</pre>

Keyword	Statement Syntax	Description	Example
assert	assert(expression) assert(expression, e	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')

### Logical Operators

You can use logical connectives for verification. In these examples, p and q represent Boolean signals or logical expressions.

Operation	Syntax	Description	Example
Negation	$\sim p$	“not p”	verify( $\sim p$ )
Conjunction	$p \ \&\& \ q$	“p and q”	verify( $p \ \&\& \ q$ )
Disjunction	$p \    \ q$	“p or q”	verify( $p \    \ q$ )
Implication	$\sim p \    \ q$	“if p, q.” Logically equivalent to implication $p \rightarrow q$ .	verify( $\sim p \    \ q$ )
Biconditional	$(p \ \&\& \ q) \    \ (\sim p \ \&\& \ \sim q)$	“p and q, or not p and not q.” Logically equivalent to biconditional $p \leftrightarrow q$ .	verify( $(p \ \&\& \ q) \    \ (\sim p \ \&\& \ \sim q)$ )

### Relational Operators

You can use relational operators for verification. 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”. 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)
```

Assigning the technically significant value of the data to a local variable inside the test sequence, and using the local variable in `verify` statements.

- Assigning the floating-point data to a fixed-point local variable, and using the local variable in `verify` statements.

Operator and Syntax	Description	Example
<code>x &gt; y</code>	Greater than	<code>verify(x &gt; y)</code>
<code>x &lt; y</code>	Less than	<code>verify(x &lt; y)</code>
<code>x &gt;= y</code>	Greater than or equal to	<code>verify(x &gt;= y)</code>
<code>x &lt;= y</code>	Less than or equal to	<code>verify(x &lt;= y)</code>
<code>x == y</code>	Equal to	<code>verify(x == y)</code>
<code>x ~= y</code>	Not equal to	<code>verify(x ~= y)</code>

## See Also

“Syntax for Test Sequences and Assessments” on page 3-32 | Test Sequence

## Related Examples

- “Test Sequence Action and Transition Operations” on page 3-11
- “Generate Function-Based Test Signals” on page 3-22

## Syntax for Test Sequences and Assessments

In this section...
“Assessment Statements” on page 3-29
“Temporal Operators” on page 3-13
“Transition Operators” on page 3-15
“Output Functions” on page 3-23
“Logical Operators” on page 3-30
“Relational Operators” on page 3-30

### Assessment Statements

You can use assessment statements inside a **Test Sequence** or **Test Assessment** block to verify simulation, stop simulation, and return verification results. 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

Keyword	Statement Syntax	Description	Example
verify	<pre>verify(expression) verify(expression, e verify(expression, i</pre>	Assesses a logical expression. Optional arguments label results in the test manager and diagnostic viewer.	<pre>verify(x &gt; y, 'SimulinkTest:great 'x and y values are %d, %d',x,y</pre>
assert	<pre>assert(expression) assert(expression, e</pre>	Evaluates a logical expression. Failure stops simulation and returns an error. Optional arguments return an error message.	<pre>assert(h == 0 &amp;&amp; k == 0,... 'h and k must initialize to 0')</pre>

## 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. Syntax in the table uses these arguments:

### TimeUnits

The units of time

Value: sec | msec | usec

Examples:

msec

### SignalCondition

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

Examples:

$u > 0$

$x \leq 1.56$

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 since the beginning of the current test step.	Before 4 seconds: before(4,sec)
duration	ElapsedTime = duration	duration returns ElapsedTime in TimeUnits since SignalCondition 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

## Transition Operators

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

Operator	Syntax	Description	Example
hasChanged	hasChanged(u)	Returns true if u changes in value since the beginning of the test	Transition when h changes: hasChanged(h)

Operator	Syntax	Description	Example
		step, otherwise returns false.  u must be an input data symbol. u cannot be an expression or other type of variable.	
hasChangedFrom	hasChangedFrom(u,A)	Returns true if u changes from the value A, otherwise returns false.  u must be an input data symbol. u cannot be an expression or other type of variable.	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. u cannot be an expression or other type of variable.	Transition when h changes to 0:  hasChangedTo(h,0)

## Output Functions

You can use functions to generate test signals. 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.

---

Function	Syntax	Description	Example
square	square(x)	Represents a square wave output with a	Output a square wave with a period of 10 sec:

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

Function	Syntax	Description	Example
latch	latch(x)	Returns the current value of x and holds that value during the current test step.	Latch b to the value of torque:  b = latch(torque)
sin	sin(x)	Returns the sine of x, where x is in radians.	A sine wave with a period of 10 sec:  sin(et*2*pi/10)
cos	cos(x)	Returns the sine 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 1/10th the test step elapsed time:  exp(et/10)

## Logical Operators

You can use logical connectives for verification. 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)

Operation	Syntax	Description	Example
Disjunction	<code>p    q</code>	“p or q”	<code>verify(p    q)</code>
Implication	<code>~p    q</code>	“if p, q.” Logically equivalent to implication $p \rightarrow q$ .	<code>verify(~p    q)</code>
Biconditional	<code>(p &amp;&amp; q)    (~p &amp;&amp; ~q)</code>	“p and q, or not p and not q.” Logically equivalent to biconditional $p \leftrightarrow q$ .	<code>verify((p &amp;&amp; q)    (~p &amp;&amp; ~q))</code>

## Relational Operators

You can use relational operators for verification. 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”. 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)
```

Assigning the technically significant value of the data to a local variable inside the test sequence, and using the local variable in `verify` statements.

- Assigning the floating-point data to a fixed-point local variable, and using the local variable in `verify` statements.

Operator and Syntax	Description	Example
<code>x &gt; y</code>	Greater than	<code>verify(x &gt; y)</code>
<code>x &lt; y</code>	Less than	<code>verify(x &lt; y)</code>
<code>x &gt;= y</code>	Greater than or equal to	<code>verify(x &gt;= y)</code>
<code>x &lt;= y</code>	Less than or equal to	<code>verify(x &lt;= y)</code>
<code>x == y</code>	Equal to	<code>verify(x == y)</code>



Operator and Syntax	Description	Example
<code>x ~= y</code>	Not equal to	<code>verify(x ~= y)</code>

### Related Examples

- “Assess Simulation Using Logical Statements” on page 3-26
- “Test Sequence Action and Transition Operations” on page 3-11
- “Generate Function-Based Test Signals” on page 3-22

## Debug a Test Sequence

### In this section...

“View Test Step Execution During Simulation” on page 3-40

“Set Breakpoints to Enable Debugging” on page 3-40

“View Data Values During Simulation” on page 3-41

“Step Through Simulation” on page 3-42


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.

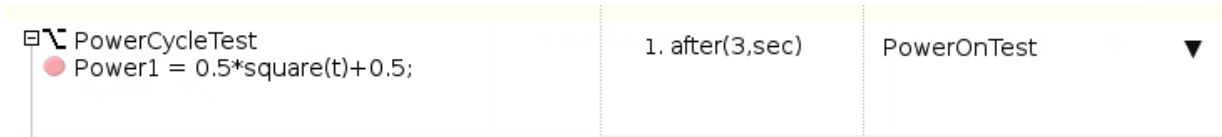
Animation speed affects simulation speed. If you slow down animation speed for debugging, return the speed to **Fast** or **Lightning Fast** when you finish debugging to avoid slowing your simulation. If you do not need the test step highlights and want the fastest simulation, choose **None**.

### 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**.



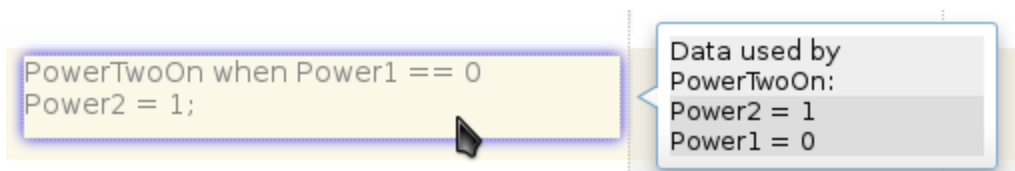
- To add a breakpoint to a test step transition, right-click the test step transition and select **Break when transition taken**.

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

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

## View Data Values During Simulation






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



**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”.

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...

“Create a Test Sequence” on page 3-43

“Simulate the Test Harness” on page 3-45

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

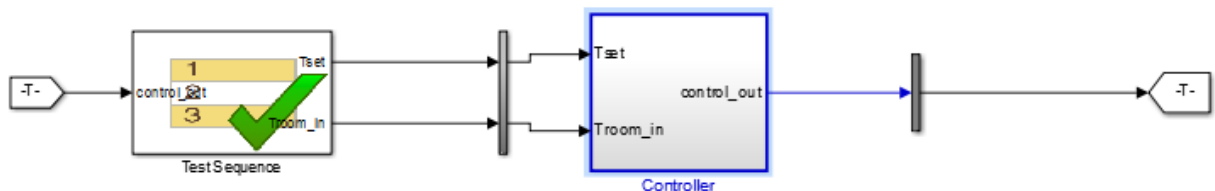
```
cd(fullfile(docroot, 'toolbox', 'sltest', 'examples'))
```

- 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 `const_90` line, and select **Add step after**. Name the third step `temp_step`.

Step	Transition	Next Step
const_90	1.	ramp_down ▼
ramp_down	1.	temp_step ▼
temp_step		

*Add step after • Add sub-step*

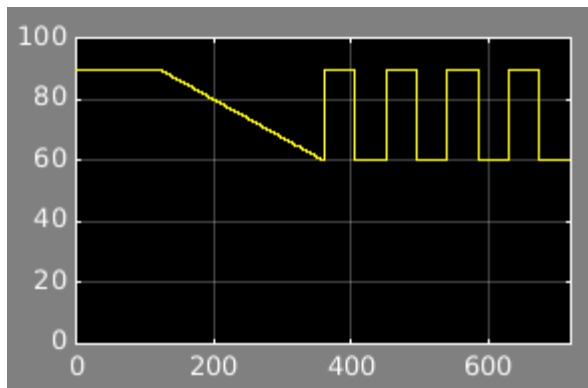
- 7 Add output conditions and transition fields to the steps. Copy and paste the listings from the table.

Step	Transition	Next step
const_90 Tset = 75; Troom_in = 90;	after(120,sec)	ramp_down
ramp_down Tset = 75; Troom_in = 90-ramp	Troom_in <= 60;	temp_step
temp_step Tset = 75; Troom_in = 75+15*sc		

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 a test sequence and test assessment for a transmission shift logic controller.

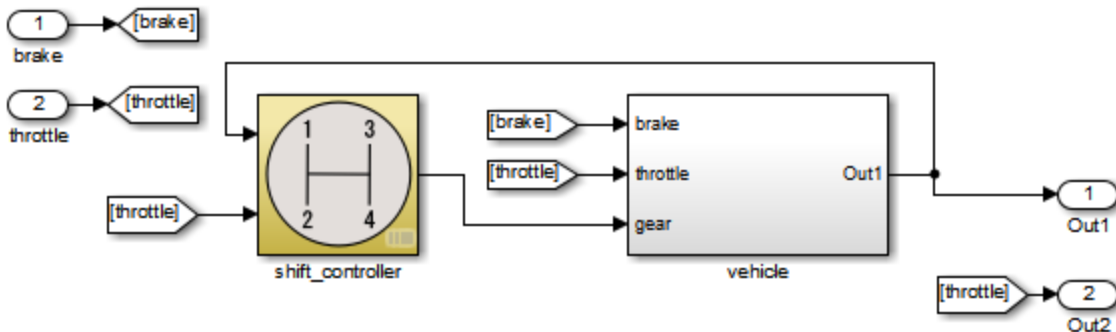
### The Model and Controller

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));
```



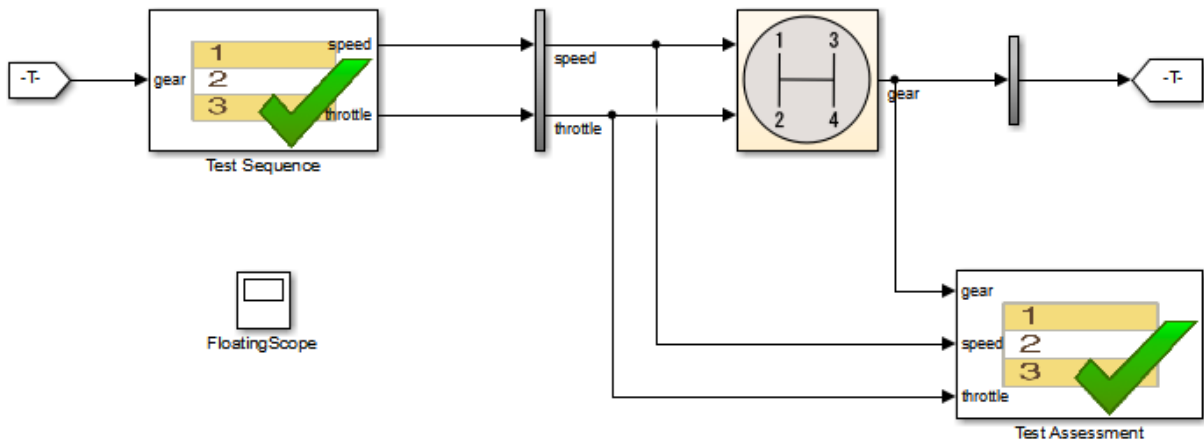
Testing Downshift Points of a Transmission Controller

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`. `shift_controller` is connected to a Test Sequence block and a Test Assessment block.

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



Copyright 2014 The MathWorks, Inc.

#### 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
<pre>initialize_4_3 throttle = 10; speed = 0+ramp(25*et);</pre>	1. speed == 75	down_4_3 ▼
<pre>down_4_3 throttle = 10+ramp(10*et); speed = 75;</pre>	1. hasChanged(gear)	initialize_3_2 ▼
<pre>initialize_3_2 throttle = 10; speed = 45;</pre>	1. after(4,sec)	down_3_2 ▼
<pre>down_3_2 throttle = 10+ramp(10*et); speed = 45;</pre>	1. hasChanged(gear)	initialize_2_1 ▼
<pre>initialize_2_1 throttle = 10; speed = 15;</pre>	1. after(4,sec)	down_2_1 ▼
<pre>down_2_1 throttle = 10+ramp(10*et); speed = 15;</pre>	1. hasChanged(gear)	stop ▼
<pre>stop throttle = 0; speed = 0;</pre>		

### 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
<div style="border: 1px solid black; padding: 5px;">           AssertConditions  <pre> assert(speed &gt;= 0, 'speed must be &gt;= 0'); assert(throttle &gt;= 0, 'throttle must be &gt;= 0 and &lt;= 100'); assert(throttle &lt;= 100, 'throttle must be &gt;= 0 and &lt;= 100'); assert(gear &gt; 0, 'gear must be &gt; 0'); </pre> </div>		
<div style="border: 1px solid black; padding: 5px;">           OverSpeed3 when gear==3            assert(speed &lt;= 90, 'Engine overspeed in gear 3')         </div>		
<div style="border: 1px solid black; padding: 5px;">           OverSpeed2 when gear==2            assert speed &lt;=50, 'Engine overspeed in gear 2')         </div>		
<div style="border: 1px solid black; padding: 5px;">           OverSpeed1 when gear==1            assert(speed &lt;= 30, 'Engine overspeed in gear 1')         </div>		
<div style="border: 1px solid black; padding: 5px;">           Else         </div>		

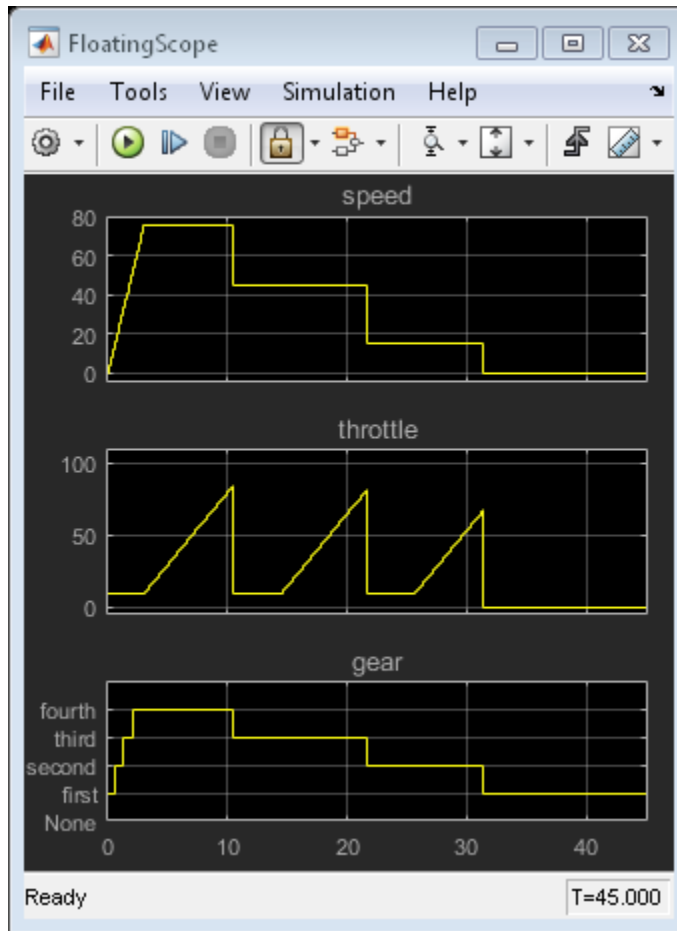
### Testing the Controller

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(md1);
```

## 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.

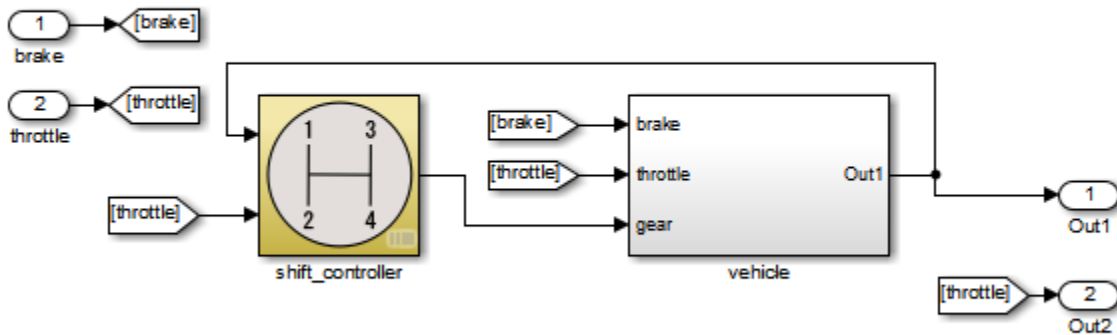
#### Explore the Test Sequence Example Model

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

sitestestSequenceExample

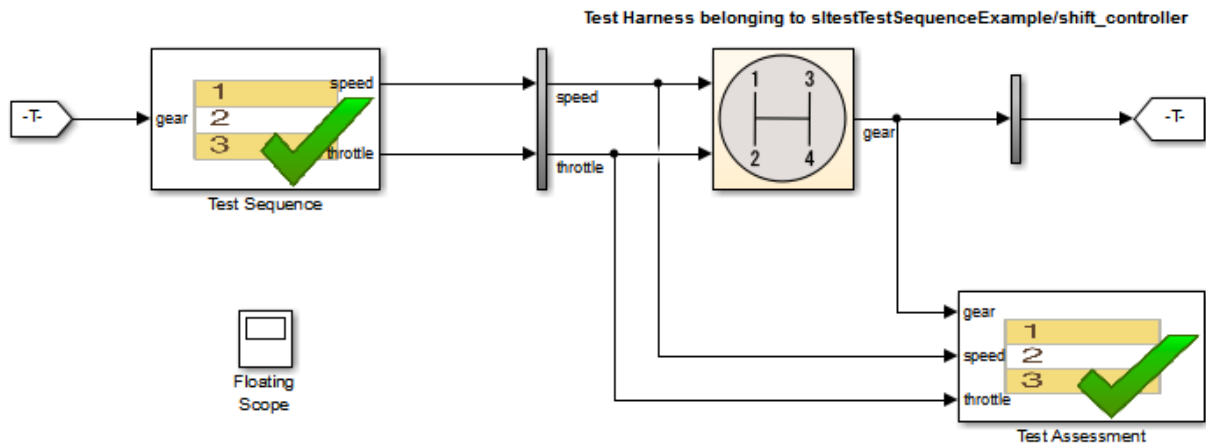
### Testing Downshift Points of a Transmission Controller

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



Copyright 2015 The MathWorks, Inc.

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



Copyright 2014 The MathWorks, Inc.



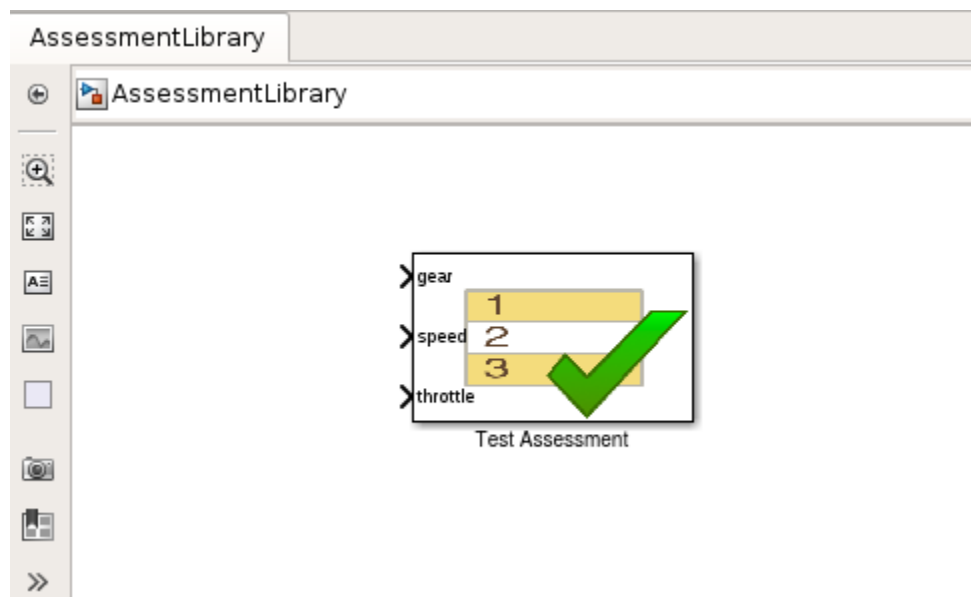


The Test Assessment block contains four assertions that define the assessment criteria:

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

### Create a Library for the Test Assessments

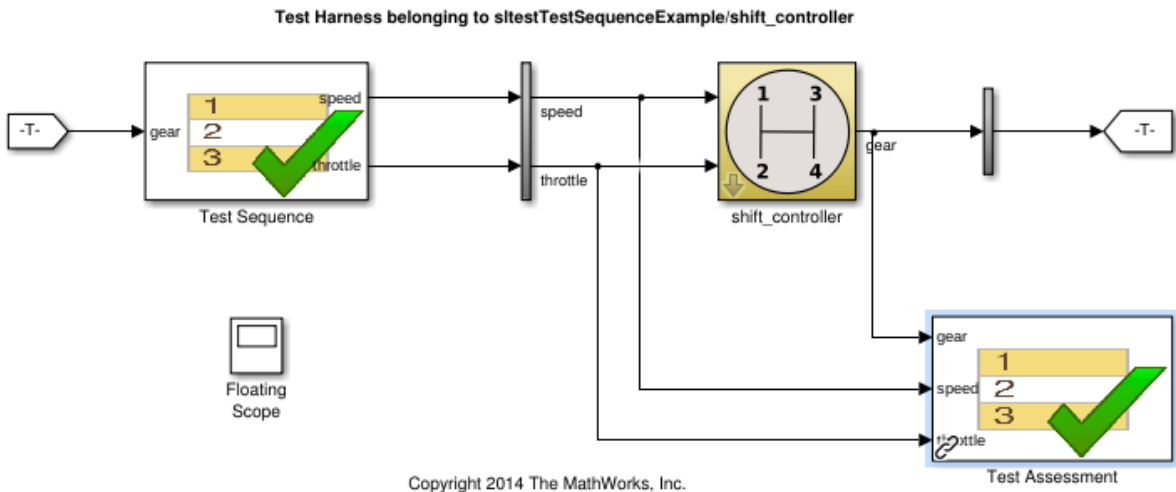
- 1 In the test harness, select **File > New > Library**.
- 2 Save the new library as **AssessmentLibrary** in a writable location on the MATLAB® path.
- 3 Copy the Test Assessment block from the test harness to the library, and then delete the Test Assessment block from the test harness.
- 4 Save the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

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

---

# SIL Verification for a Subsystem

<b>In this section...</b>
“Create a SIL Verification Harness for a Controller” on page 4-3
“Configure and Simulate a SIL Verification Harness” on page 4-5
“Compare the SIL Block and Model Controller Outputs” on page 4-5

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 SIL and PIL Simulations”.

Also see “Code Generation of Subsystems” in the Simulink Coder™ documentation.

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

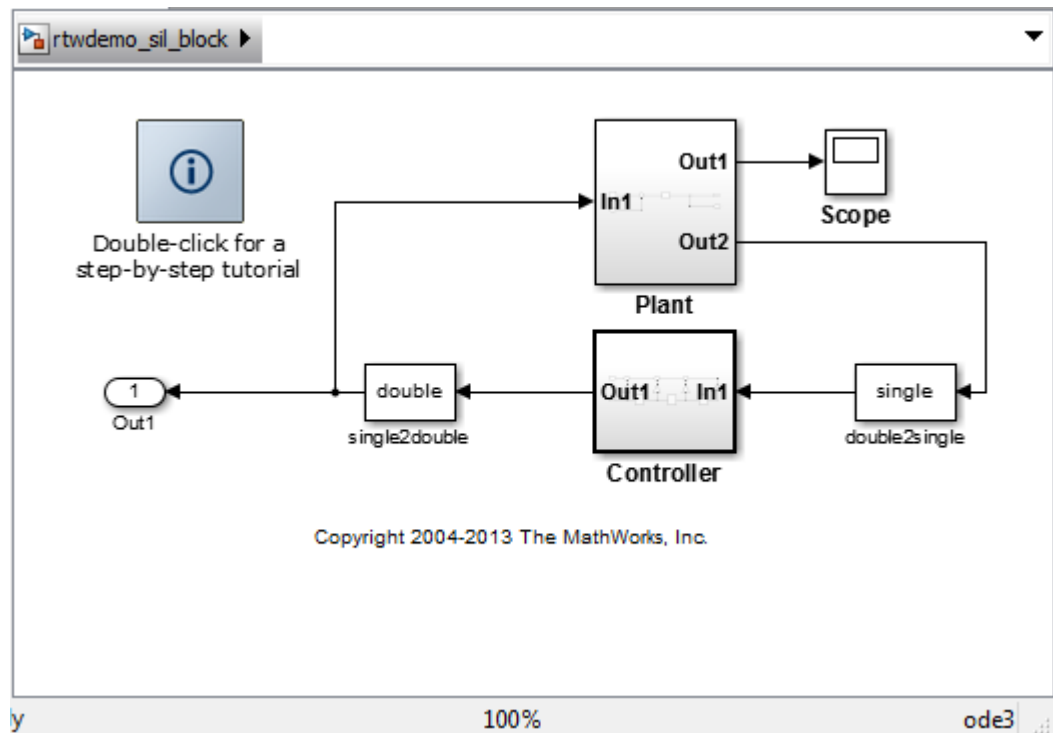
## Create a SIL Verification Harness for a Controller

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,



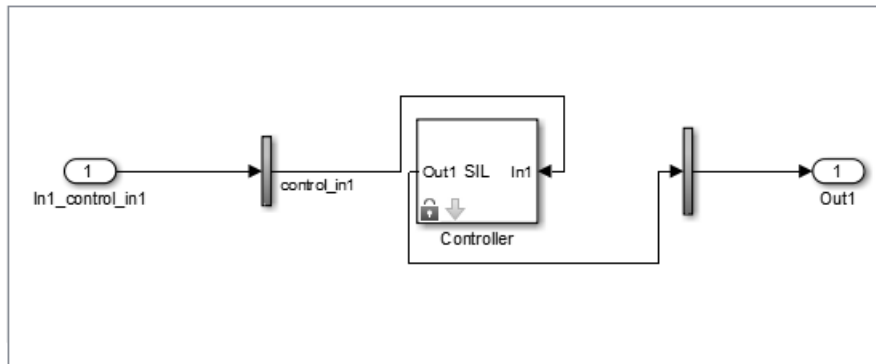
- 2 Save a copy of the model using the name `controller_model` in a new folder, in a writable location on the MATLAB path.
- 3 Enable signal logging for the model. At the command prompt, enter

```
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
 

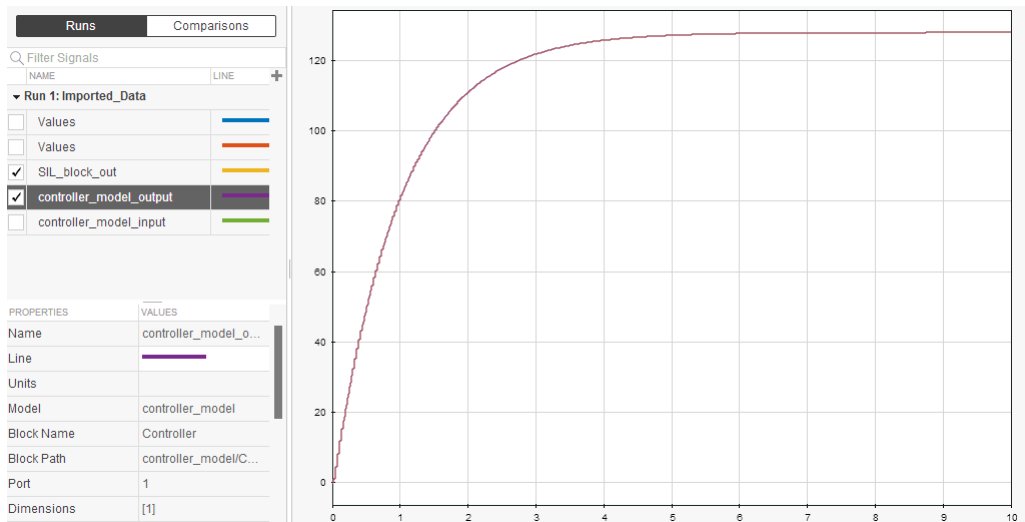
```
set_param('SIL_harness', 'SignalLogging', 'on', 'SignalLoggingName', ...
'harness_signals', 'SignalLoggingSaveFormat', 'Dataset')
```
- 3 Right-click the output signal of the SIL block and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `SIL_block_out`. 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 Signal Data from Multiple Simulations” in the Simulink documentation.



- 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.



# Simulink Test Manager Introduction

---

## Introduction to the Test Manager

### In this section...

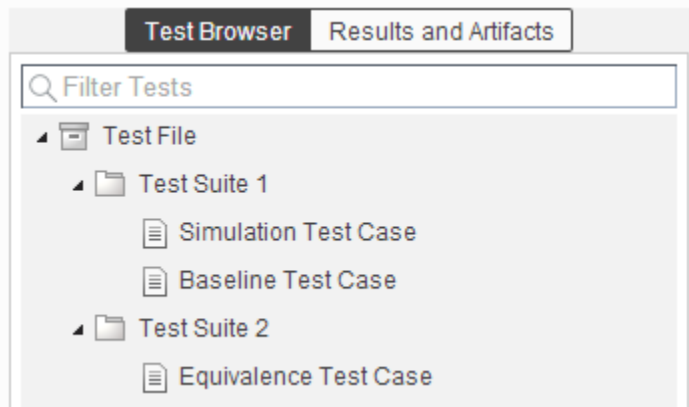
- “Test Manager Description” on page 5-2
- “Test Creation and Hierarchy” on page 5-2
- “Test Results” on page 5-3
- “Share Results” on page 5-3

### 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 Two Simulations for Equivalence”
- “Code Generation Verification Workflow with Simulink Test”



# Test Manager Test Cases

---

- “Manage Test File Dependencies” on page 6-2
- “Test Model Output Against a Baseline” on page 6-9
- “Test a Simulation for Run-Time Errors” on page 6-13
- “Generate Test Cases from Model Components” on page 6-16
- “Use External Inputs in Test Cases” on page 6-24
- “Automate Tests Programmatically” on page 6-27
- “Run Multiple Combinations of Tests Using Iterations” on page 6-30
- “Collect Coverage in Tests” on page 6-38
- “Run Tests Using Parallel Execution” on page 6-44
- “How Tolerances Are Applied to Test Criteria” on page 6-46
- “Test Manager Limitations” on page 6-47
- “Test Sections” on page 6-49
- “Test Models Using Inputs Generated by Simulink Design Verifier” on page 6-57

## Manage Test File Dependencies

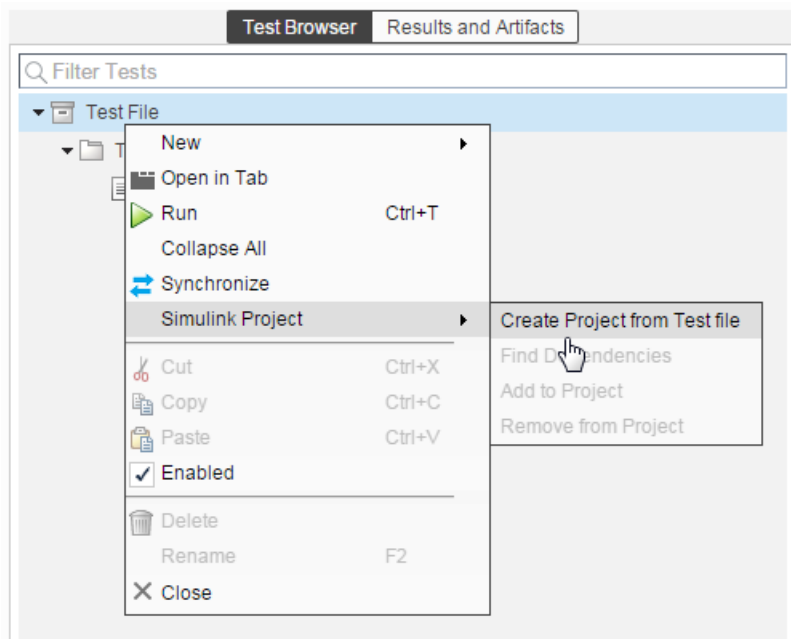
In this section...
“Package a Test File Using Simulink Projects” on page 6-2
“Find Test File Dependencies and Impact” on page 6-4
“Share a Test File with Dependencies” on page 6-8

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 all the 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.

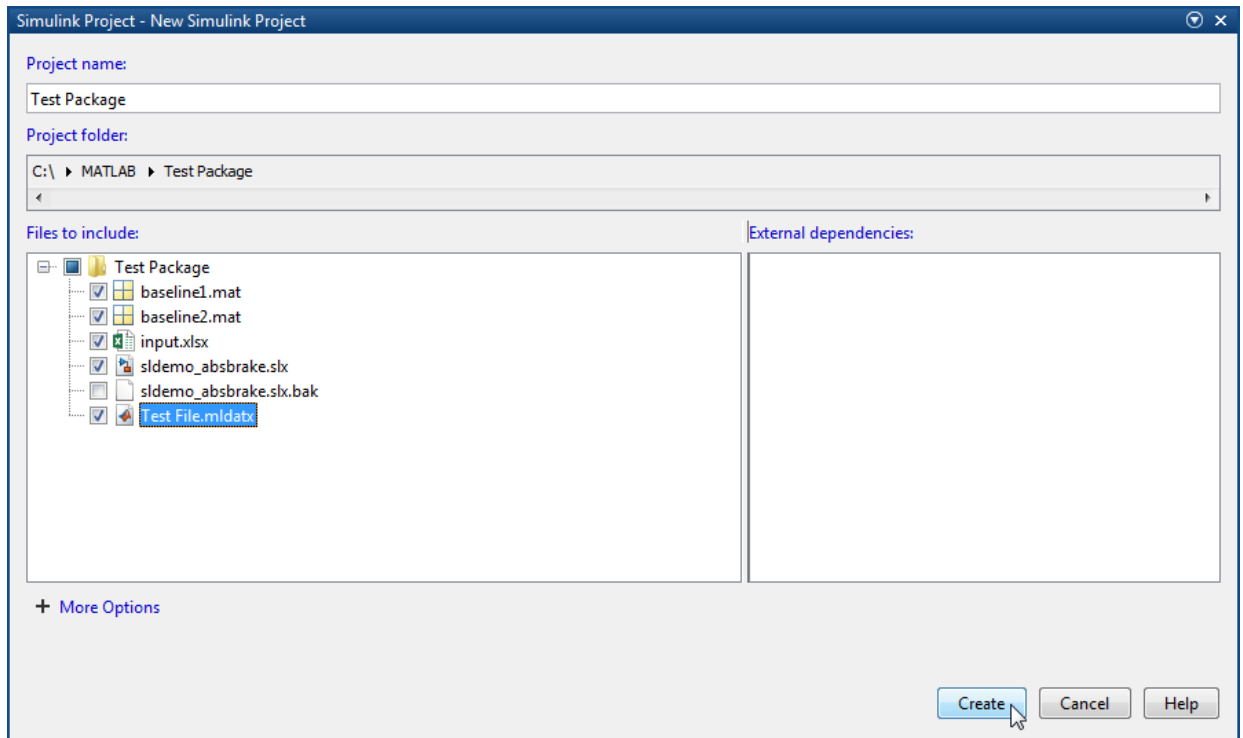
### Package a Test File Using Simulink Projects

- 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**.

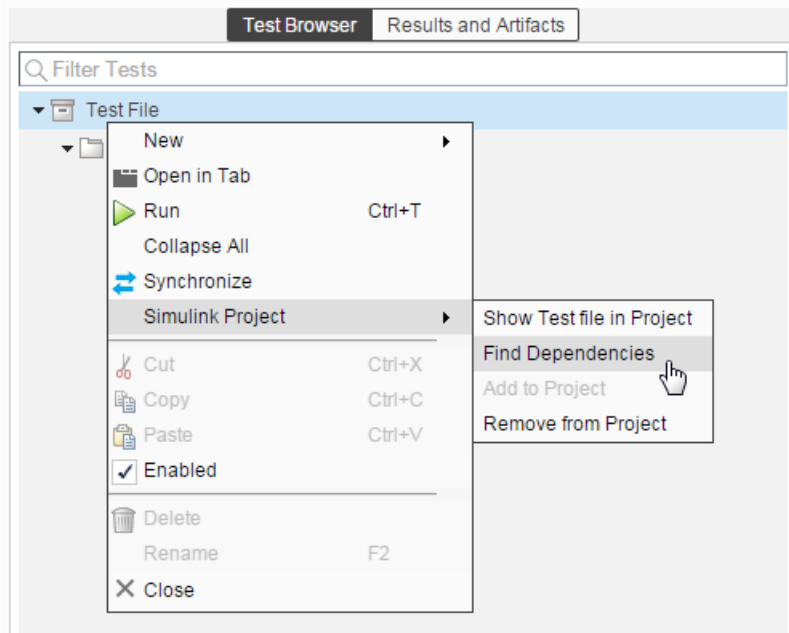


### Find Test File Dependencies and Impact

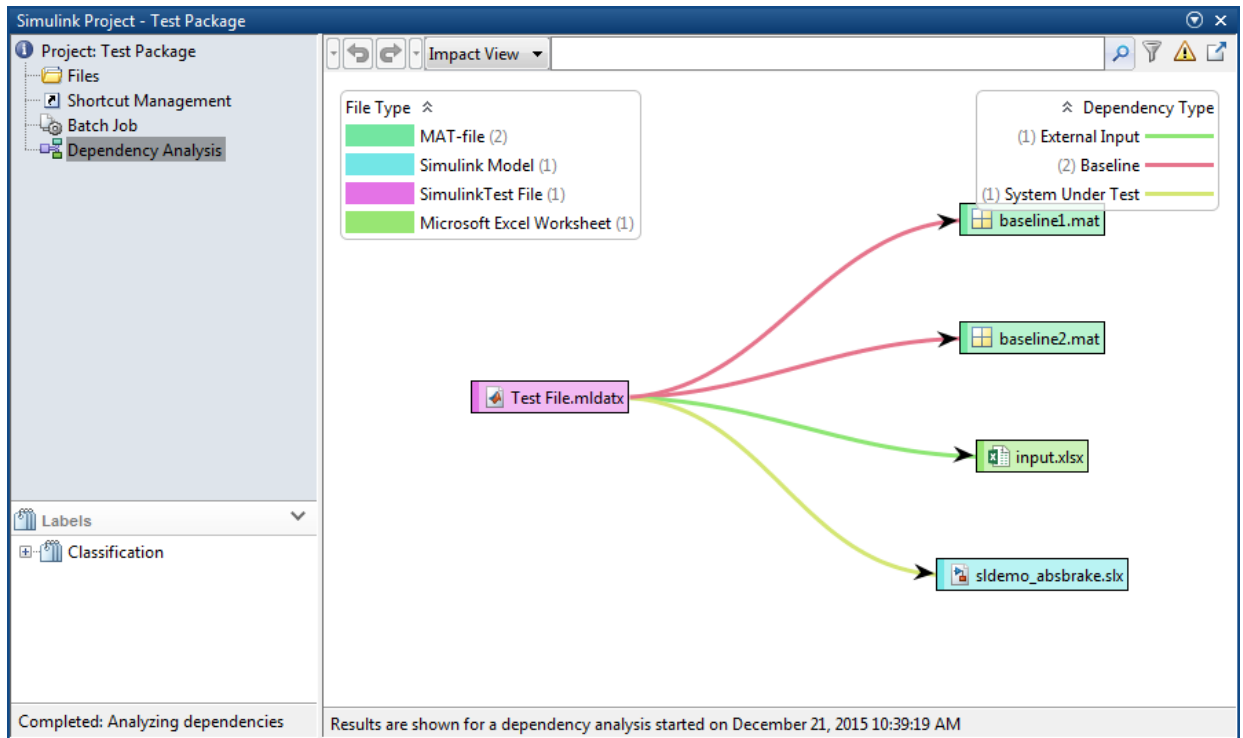
If you have a test file saved in a Simulink project, then you can find the file dependencies.

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



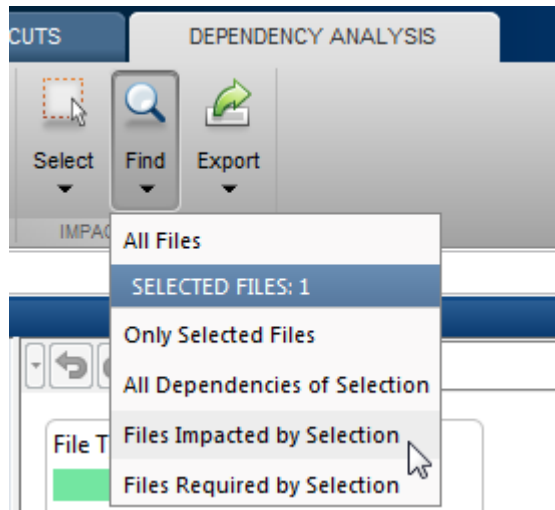


Simulink Projects opens and shows a graph of file dependencies.

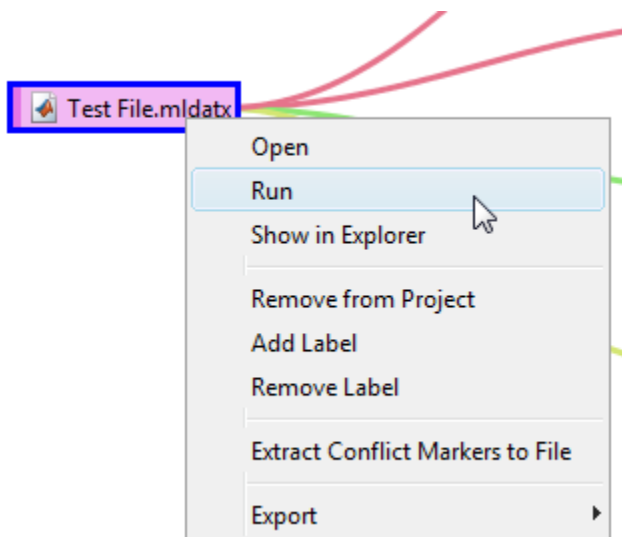


If you want to change a model or requirement, then you can find the impact that the change could have on testing.

- 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?”


## Test Model Output Against a Baseline

To test the simulation output of a model against a defined baseline data set, 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 **Baseline Criteria**, click **Capture** to record a baseline data set from the model specified under **System Under Test**.

Save the baseline data set 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 data set. Set the **Absolute Tolerance** of the first `yout` signal to 15, which corresponds to the `Ww` signal.

SIGNAL NAME	ABS TOL	REL TOL
test_capture.mat	0	0.00%
yout	15	0.00%
yout	0	0.00%
yout	0	0.00%
slp	0	0.00%

+ Add... Capture... Delete

To add or remove columns in the baseline criteria table, click the column selector button . For more information about tolerances and criteria, see “How Tolerances Are Applied to Test Criteria” on page 6-46.

## 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 toolbar, 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.

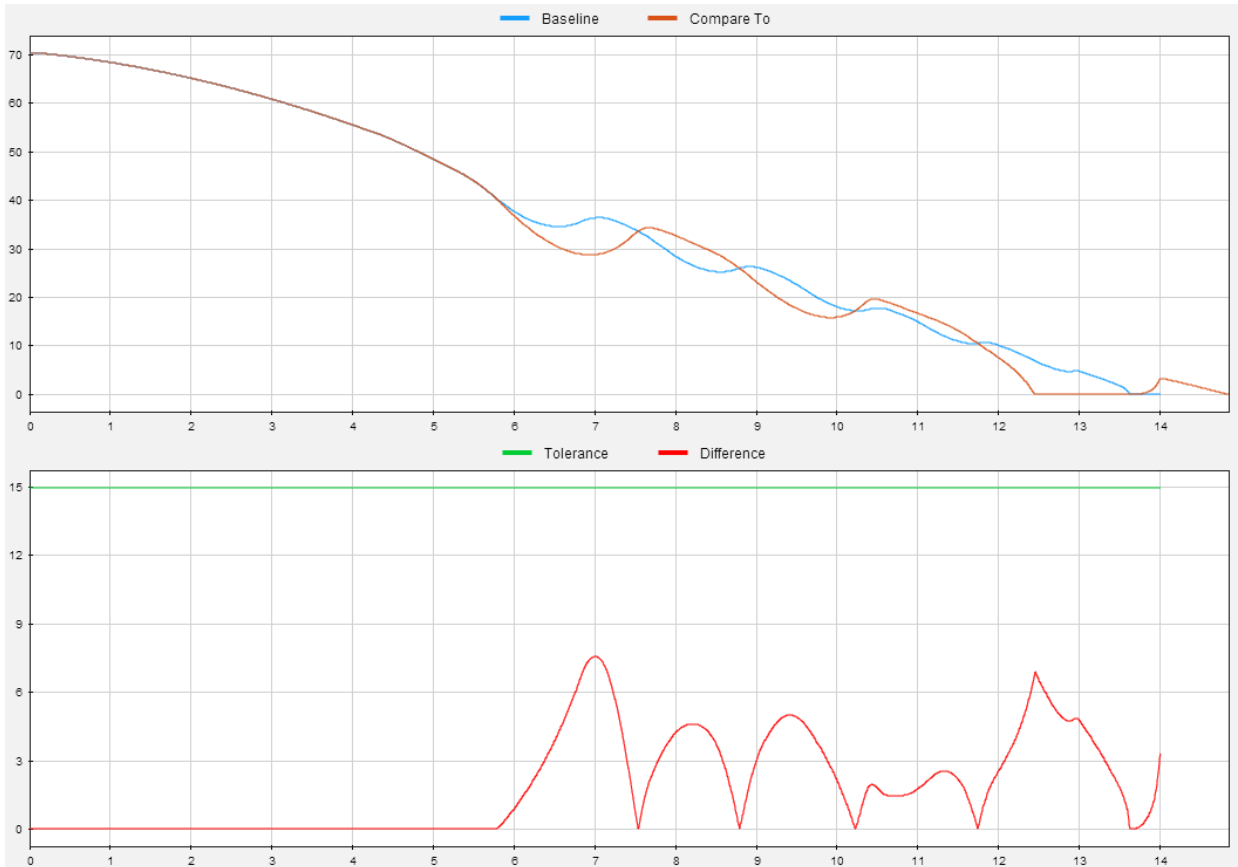
- 4 Expand the results until you see the baseline criteria result.

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.

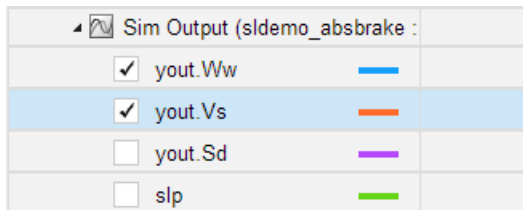
- 5 To view the `yout.Ww` signal comparison between the model and the baseline criteria, expand **Baseline Criteria Result** and click the option button next to the `yout.Ww` signal.

  Baseline Criteria Result	
<input checked="" type="radio"/> <code>yout.Ww</code>	
<input type="radio"/> <code>yout.Vs</code>	
<input type="radio"/> <code>yout.Sd</code>	
<input type="radio"/> <code>slp</code>	
  Sim Output (sldemo_absbrake :	

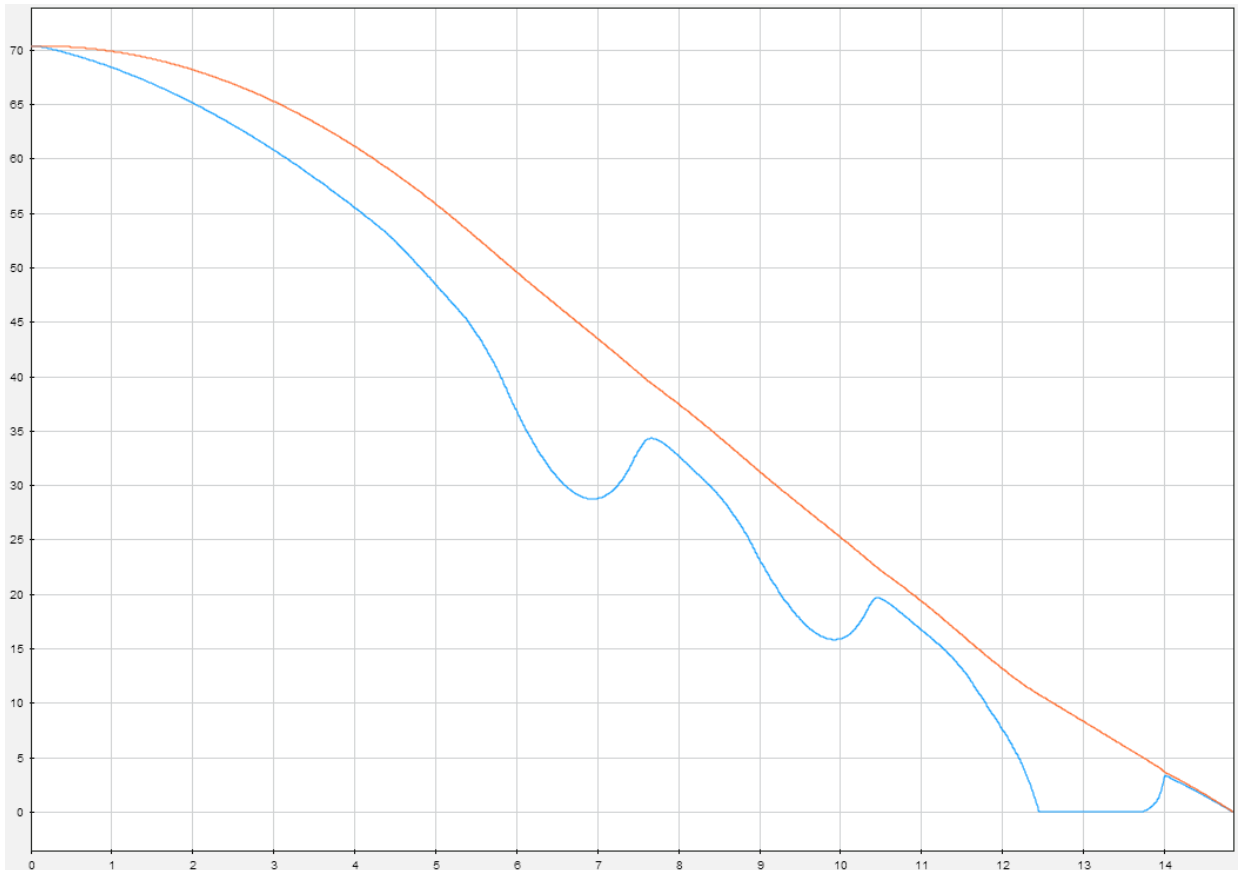
The **Comparison** tab opens and shows the criteria comparisons for the `yout.Ww` signal.



- 6 You can also view signal data from the simulation. Expand **Sim Output** and select the signals you want to plot.



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.



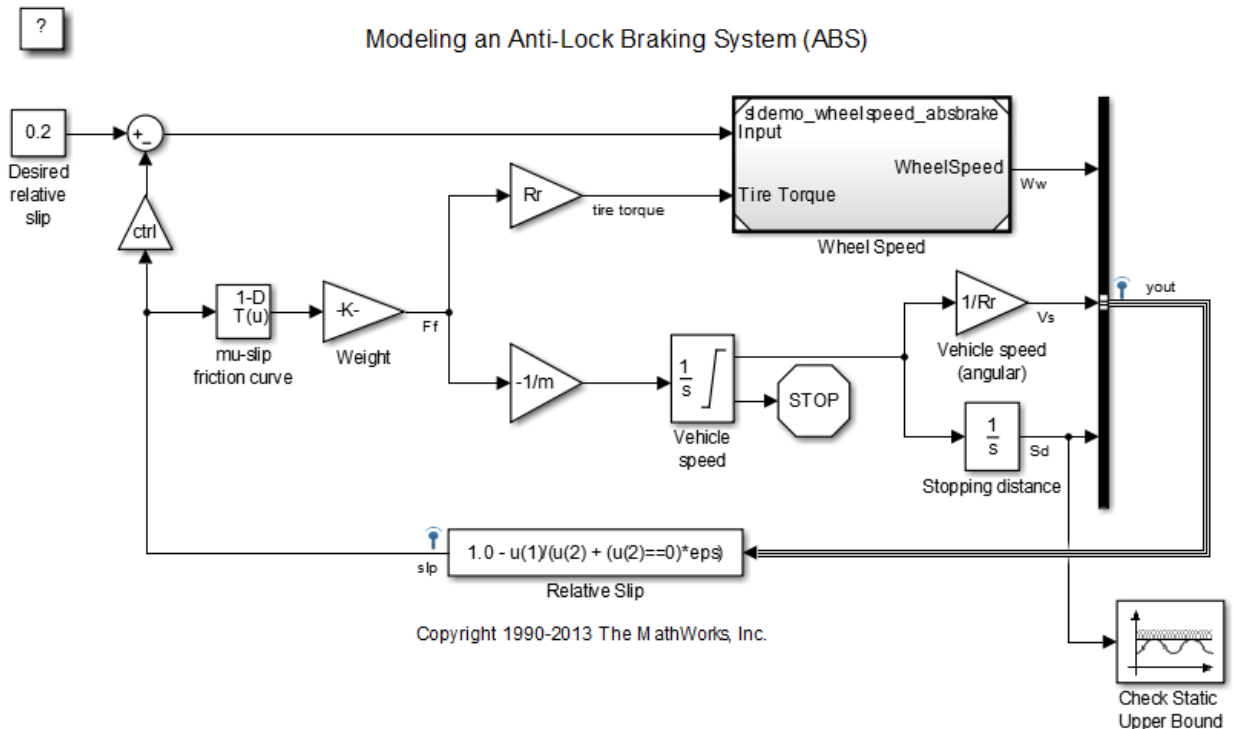
## Test a Simulation for Run-Time Errors

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 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.
- 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
<input checked="" type="checkbox"/> Parameter Set 1		
<input checked="" type="checkbox"/> 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 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.

## View Test Results

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

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

- 2 The result shows a red X, which indicates a test failure. In this case, the model stopping distance exceeded the upper bound of 725 and triggered an assertion from the Check Static Upper Bound block.

▼ SUMMARY	
Name	<a href="#">Upper Bound Test</a>
Outcome	✘

Look under **Errors** for the details of the assertion failure.

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

## Generate Test Cases from Model Components

In this section...
“Generate the Test Cases” on page 6-16
“Synchronize Test Cases” on page 6-18
“Generate Test for a Subsystem” on page 6-20

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:

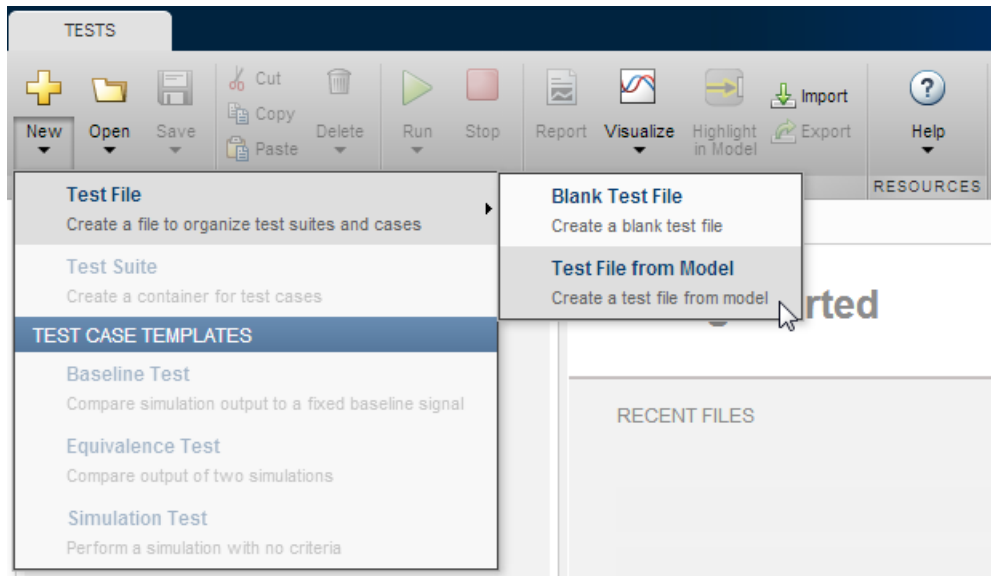
- **Signal Builder** block in the top model
- Test harnesses from the top model or any subsystem
- **Signal Builder** block at the top level of a test harness

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.

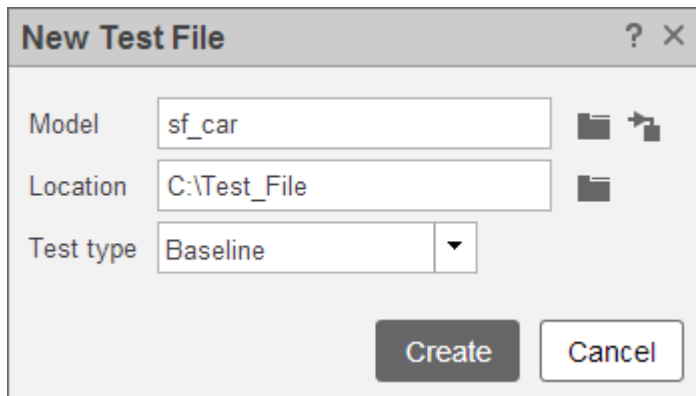
If you want to generate a test case for a model subsystem using a harness, see “Generate Test for a Subsystem” on page 6-20.

### 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 all test cases.

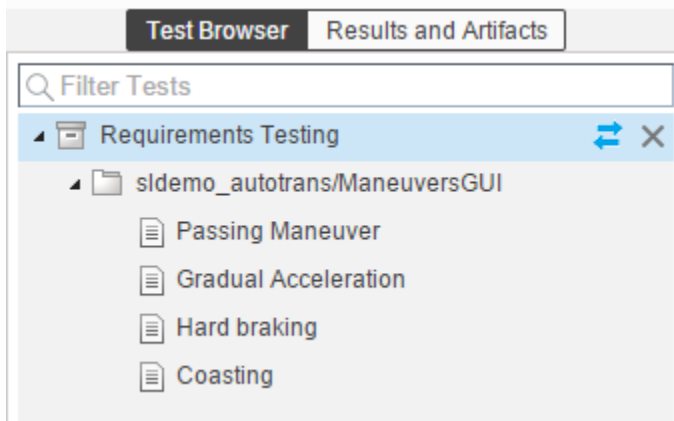


- 4 Click **Create**.


## Synchronize Test Cases

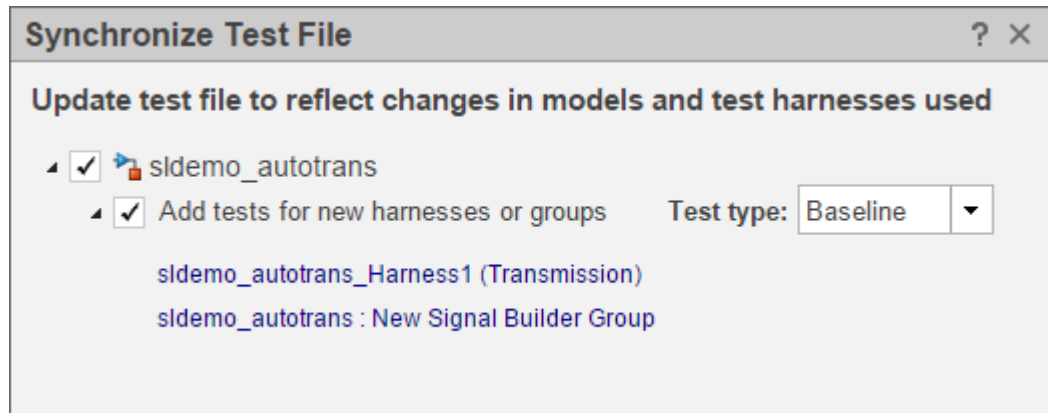
If you add model components to your model, such as **Signal Builder** groups or test harnesses, then you can generate new test cases in the test manager to synchronize your model. Also, if you remove model components, then you can disable or delete test cases in the test manager when you synchronize. 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 generate test cases from the model using **New > Test File > Test File from Model**, then test cases generate using the **Signal Builder** groups.



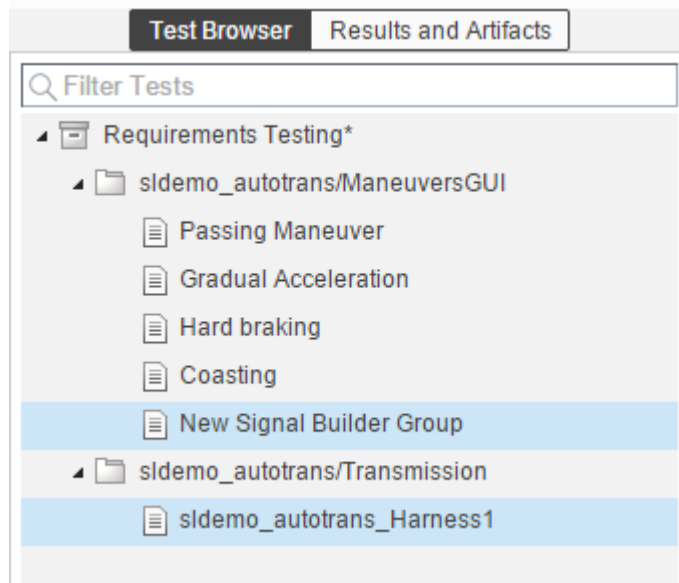
If you add another **Signal Builder** group, **New Signal Builder Group**, and a test harness, `sldemo_autotrans_Harness1`, then you can add test cases for these model components. Synchronize the model and test file.

- 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 Review the synchronization dialog box to add or remove any test cases, and select the test case type: baseline, equivalence, or simulation.



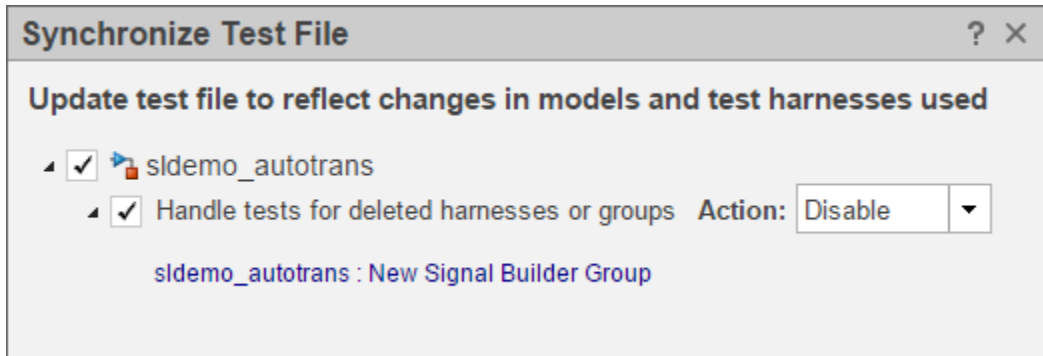
- 4 To complete the synchronization, click **Update Test File**.

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



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.



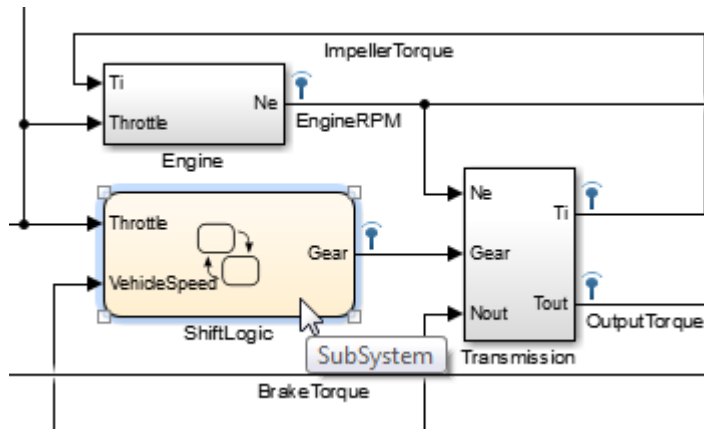
## Generate Test for a Subsystem

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.

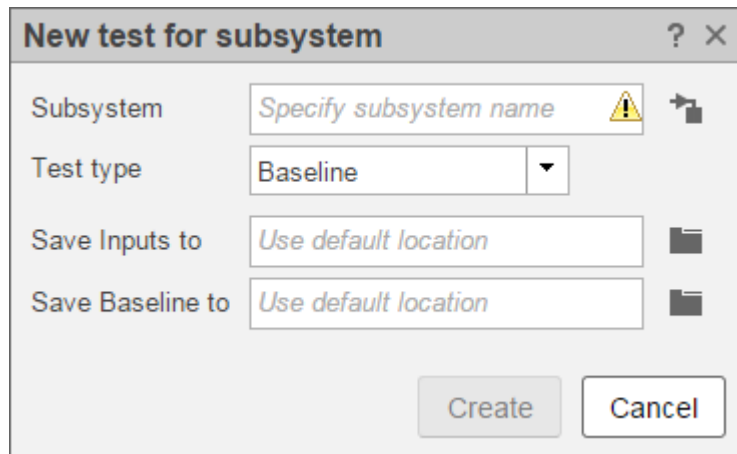
To create a test from a subsystem:


- 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 toolbar, click the **New** arrow and select **Test for Subsystem**. A dialog box opens to help you create the test.



- 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.

- 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 generated for you.

▼ SYSTEM UNDER TEST

Model:

▼ TEST HARNESS

Harness:

▼ INPUTS

EXTERNAL INPUTS

NAME	FILE	SHEET	STATUS
<input checked="" type="checkbox"/> inputs	C:\MATLAB\inputs		<a href="#">Mapped</a>

▼ BASELINE CRITERIA

Save baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL	
<input type="checkbox"/> 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 as inputs in a test case. External inputs are mapped to the model using root inport mapping under 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 root inport mapping modes, supported data types or formats, and mapping results, see “Map Root Inport Signal Data”.

### 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”.
- 5 Click **Map Inputs**. The **Mapping Status** table shows if the port and signals map successfully.

For more information about troubleshooting the mapping status, see “Understand Mapping Results”.

- 6 Click **Apply**.

### Use Microsoft Excel File for Inputs

The Root Inport Mapping 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 all columns do not have signal names, the tool assigns signal names using the format **Signal#**, where # increments with each additional signal.
- All 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** ? X

**INPUT FILE SPECIFICATION**

File: ..\CalibrationSet.xlsx

Sheet: OxygenSensorWarmup

Create scenarios from each sheet

▼ **INPUT MAPPING**

Mapping Mode: Port Order Map Inputs

▼ **MAPPING STATUS**

Successfully mapped inputs.

PORT	BLOCK NAME	MAPPED SIGNAL	STATUS
1	inout/In1		✓

► **ADVANCED**

Apply Cancel

- 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”.
- 7 Click **Map Inputs**. The **Mapping Status** table shows if the port and signals map successfully.

For more information about troubleshooting the mapping status, see “Understand Mapping Results”.

- 8 Click **Apply**.

EXTERNAL INPUTS

NAME	FILE	SHEET	STATUS
<input checked="" type="checkbox"/> CalibrationSet.xlsx	C:\MATLAB\CalibrationSet.xlsx	OxygenSensorWarmup	Mapped

Signal Builder Group [Model Settings] + Add ✎ Edit ↻ Refresh 🗑️ Delete

## Automate Tests Programmatically

### In this section...

“List of Functions and Classes” on page 6-27

“Create and Run a Test Case” on page 6-28

### List of Functions and Classes

Function	Description
<code>sltest.testmanager.view</code>	Launch the Simulink Test manager
<code>sltest.testmanager.createTestsFrom</code>	Generate test cases from a model
<code>sltest.import.sldvData</code>	Create test cases from Simulink Design Verifier results
<code>sltest.testmanager.load</code>	Load a test file in the Simulink Test manager
<code>sltest.testmanager.run</code>	Run all test files in the Simulink Test manager
<code>sltest.testmanager.copyTests</code>	Copy test cases or test suites to another location
<code>sltest.testmanager.moveTests</code>	Move test cases or test suites to a new location
<code>sltest.testmanager.report</code>	Generate report of test results
<code>sltest.testmanager.clear</code>	Clear all test files from the Simulink Test manager
<code>sltest.testmanager.close</code>	Close the Simulink Test manager
<code>sltest.testmanager.clearResults</code>	Clear all results from the Simulink Test manager
<code>sltest.testmanager.importResults</code>	Import test manager results file
<code>sltest.testmanager.exportResults</code>	Export results set from test manager
<code>sltest.testmanager.getResultSets</code>	Returns result set objects in test manager
Class	Description
<code>sltest.testmanager.TestFile</code>	Create or modify test file

Class	Description
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.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 Test Case

This example shows how to use the `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 data set.

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 a signal.

```
sc = getSignalCriteria(baseline);  
sc(1).AbsTol = 9;
```

Run the test case and return an object with results data. The results set object gives information about the number of passed, failed, and disabled test cases.

```
ResultsObj = run(tc);
```

Open the test manager so you can view the simulation output and comparison data.

```
sltest.testmanager.view;
```

The test case fails because only one of the signal comparisons between the simulation output and the baseline criteria is within tolerance.

Generate a report from the results data.

```
filePath = 'test_report.pdf';  
sltest.testmanager.report(ResultsObj, filePath, 'Author', 'Test Engineer', ...  
    'IncludeSimulationSignalPlots', true, 'IncludeComparisonSignalPlots', true);
```

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.

## See Also

`sltest.testmanager.report`

## Run Multiple Combinations of Tests Using Iterations

### In this section...

“Create Table Iterations” on page 6-30

“Create Scripted Iterations” on page 6-33

“Sweep Through a Set of Parameters” on page 6-36

Test manager iterations facilitate test cases for multiple data sets. Use iterations to test many different combinations of parameter sets, external inputs, configuration sets, **Signal Builder** groups, or baseline data sets. 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:


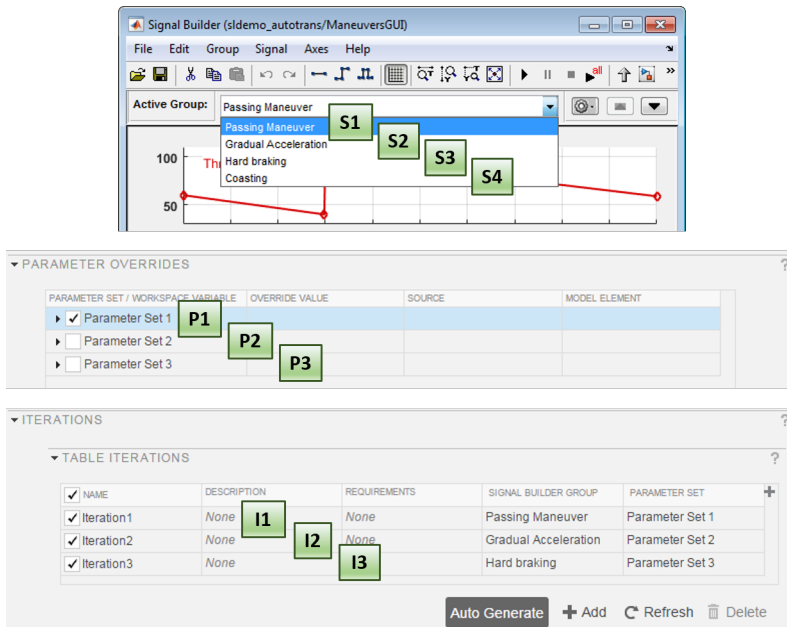
- 1 Add parameter sets, external inputs, configuration sets, **Signal Builder** groups, or baselines to a test case if they are applicable to your tests.
- 2 To add an iteration to the table manually, click **Add**.
- 3 By default, the **Parameter Set** and **External Input** columns are visible in the 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

<input checked="" type="checkbox"/> NAME	DESCRIPTION	REQUIREMENTS	SIGNAL BUILDER GROUP	PARAMETER SET
<input checked="" type="checkbox"/> Iteration	None	None	[Default] None	[Default]

Auto Generate + Add Delete

Auto-generated 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. There are three iterations because 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.



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 the Table Iterations That Will Run

To see a list of all of the iterations that will run from the table iterations section, click **Show Iterations**. The test manager generates a list of iterations that will run. 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, then the test manager combines iterations and lockstep ordering applies.

Section Option	Purpose
Signal Builder Group	Applies to the <b>Inputs</b> section of a simulation, baseline, or equivalence test case, for the specified <b>Signal Builder</b>

Section Option	Purpose
	<b>Group.</b> Each Signal Builder group is used to generate an iteration.
Parameter Set	Applies to the <b>Parameter Overrides</b> section of a simulation, baseline, or equivalence test case. Each parameter override set is used to generate an iteration.
External Input	Applies to the <b>Inputs</b> section of a simulation, baseline, or equivalence test case, for the specified <b>External Inputs</b> data sets. Each external input set is used to generate an iteration.
Configuration Set	Applies to the <b>Configuration Setting Overrides</b> 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 <b>Baseline Criteria</b> section of a baseline test case. Each baseline criteria data 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, there is a menu for <b>Simulation 1</b> or <b>Simulation 2</b> . These 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 runtime when a test executes. Enter the script into the Scripted Iterations section text box.

▼ ITERATIONS

▶ TABLE ITERATIONS

▼ SCRIPTED ITERATIONS

▶ Help on creating test iterations:

```
1 % 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 a `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 6-30.

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 all the iterations that execute when you run the test case, click **Show Iterations**.

### 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

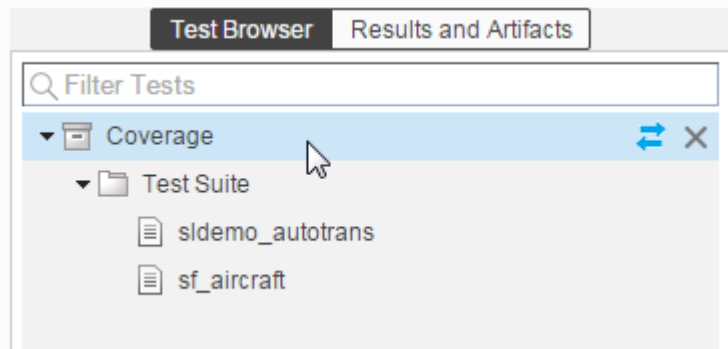
## Collect Coverage in Tests

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.

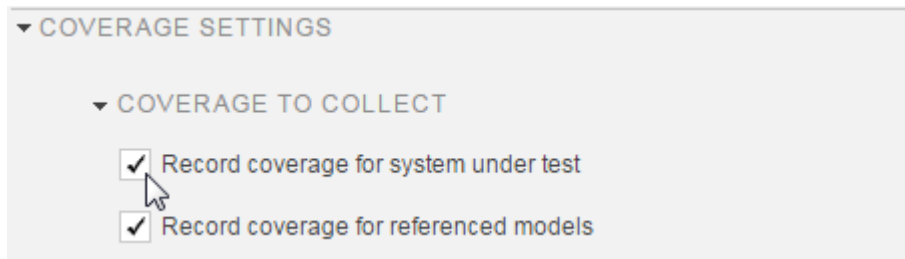
For example, to test a model for coverage, turn on coverage at the test-file, test-suite, or the individual test-case level. If you set coverage settings at the test-file level, then the test suite and test cases in the test file inherit the coverage settings. At the test case and test-suite level, you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

To turn on and collect coverage at a test-file level:

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

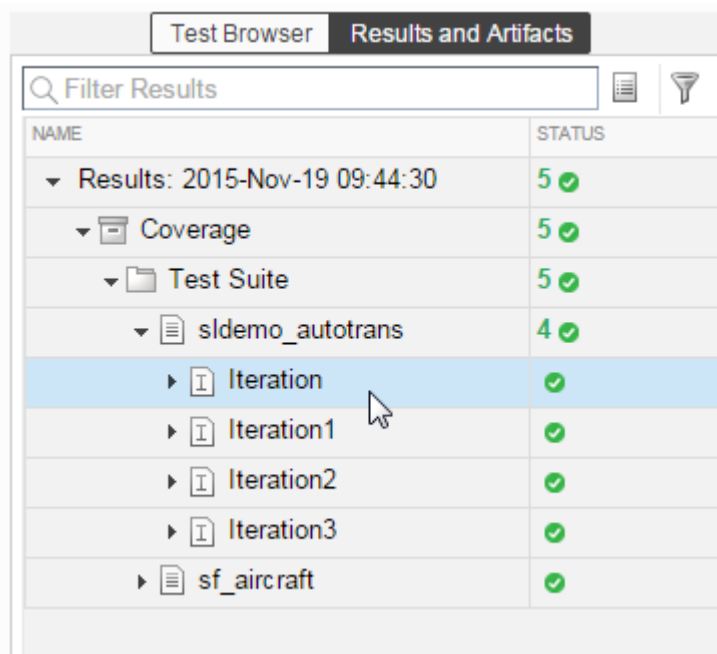


- 2 Expand the **Coverage Settings** section in the test file and select **Record coverage for system under test**. If you have referenced model that you want to collect coverage for, then select **Record coverage for referenced models**.

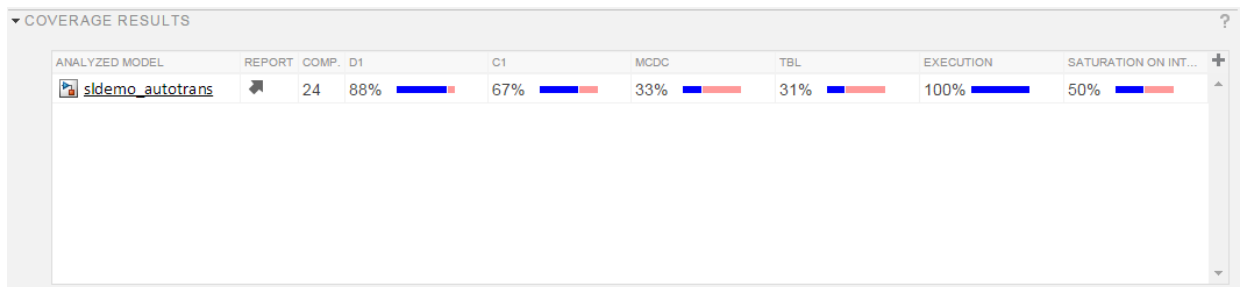


- 3 Select the coverage metrics you want to collect when the test cases in the test file execute. For more information about the types of model coverage, see “Types of Model Coverage”.
- 4 Run the test file.
- 5 To view the collected coverage results, select a test case result in the **Results and Artifacts** pane.

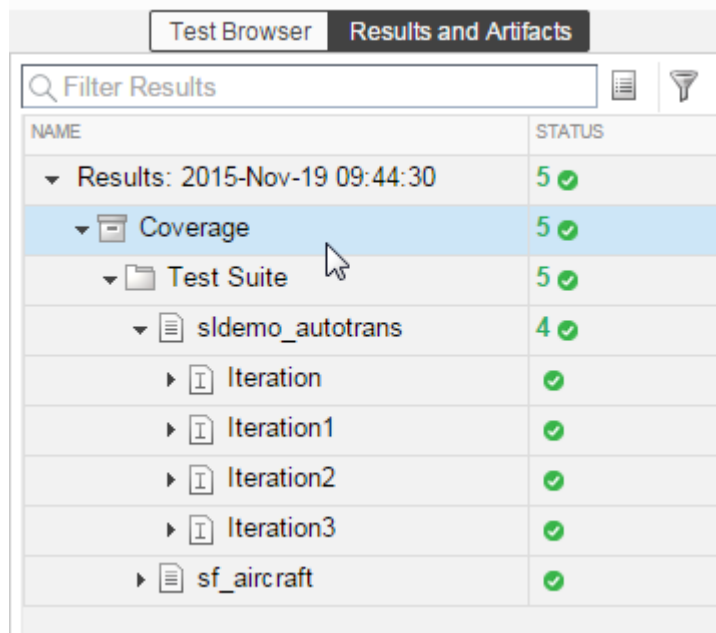
In this example, there is a test case that contains iterations for a Signal Builder block. Each iteration has coverage results.



- To view the coverage data, expand the **Coverage Results** section of the test case result.



- Select the result to view the aggregated coverage for a test case, test suite, or test file.



- 8 To view the coverage data, 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.

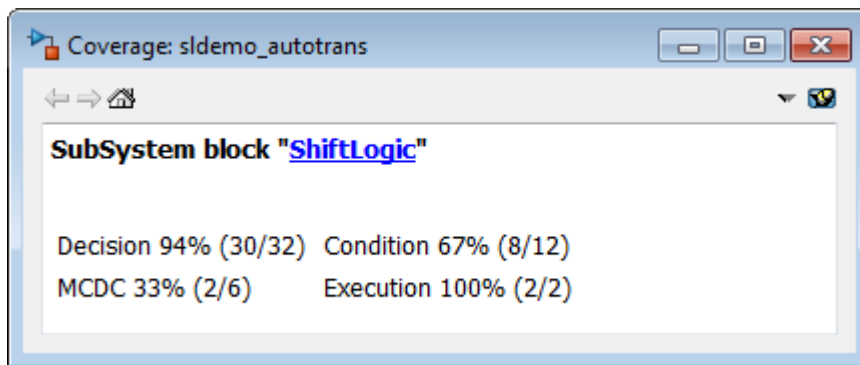
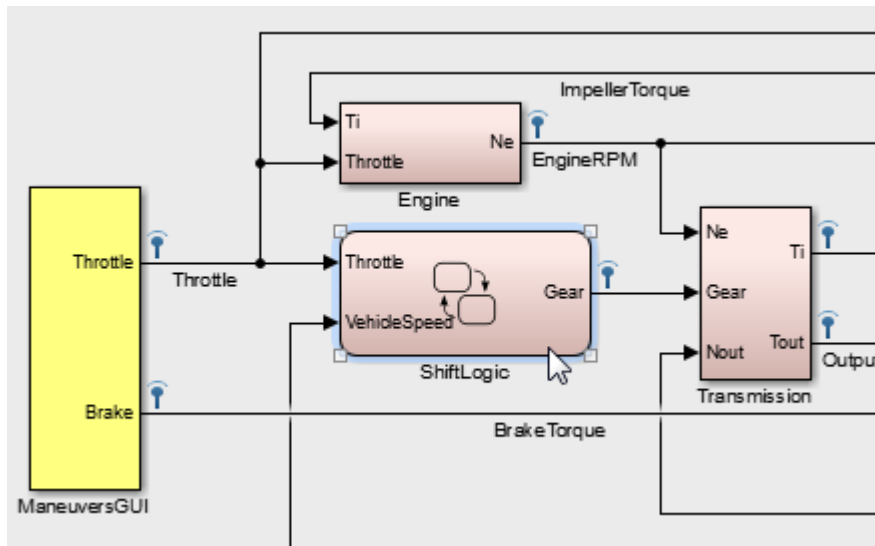
▼ AGGREGATED COVERAGE RESULTS

ANALYZED MODEL	REPORT	COMP.	D1	C1	MCDC	TBL	EXECUTION	SATURATION ON INT...
<a href="#">sf_aircraft</a>	📄	428	47%	40%	12%	33%	98%	50%
<a href="#">sldemo_autotrans</a>	📄	24	94%	67%	33%	44%	100%	50%

- 9 To view the coverage results graphically in the model, select 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.



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

**See Also**

“Specify Model Coverage Options”

## Run Tests Using Parallel Execution

### In this section...

“Use Parallel Execution” on page 6-44

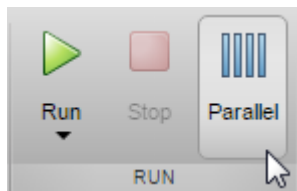
“When Will Tests Benefit from Using Parallel Execution?” on page 6-45

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 “Clusters and Cluster Profiles”.
- 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 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 “Parallel Preferences”.
- Load Simulink on all of the parallel pool workers.



## When Will Tests Benefit from Using Parallel Execution?

In general, there are a few factors that could indicate that using parallel execution would result in faster tests. A complex Simulink model that takes a long time to simulate could benefit from parallel execution. Also, parallel execution could help if you have a large number of long-running tests, such as iterations.

### See Also

`sltest.testmanager.run`

### Related Examples

- “Clusters and Clouds”

## How Tolerances Are Applied to Test Criteria

Tolerances can be specified in the **Baseline Criteria** or **Equivalence Criteria** sections of test cases. The default value for the relative tolerance and absolute tolerance for a signal comparison is zero. If you specify tolerances, then the test calculates the tolerances as follows:

$$\text{tolerance} = \max(\text{absoluteTolerance}, \text{relativeTolerance} * \text{abs}(\text{baselineData}));$$

The more lenient tolerance is used to determine the pass-fail outcome of the criteria comparison.

### Modify Criteria Tolerances

You can change the criteria tolerances in the **Baseline Criteria** or **Equivalence Criteria** sections of baseline or equivalence test cases, respectively. To modify a tolerance, select the signal name in the criteria table and double-click the tolerance value.

▼ BASELINE CRITERIA

SIGNAL NAME	ABS TOL	REL TOL
<input checked="" type="checkbox"/> baseline_data.mat	0	0.00%
<input checked="" type="checkbox"/> yout	0	0.00%
<input checked="" type="checkbox"/> yout	0	0.00%
<input checked="" type="checkbox"/> yout	0	0.00%
<input checked="" type="checkbox"/> slp	0	0.00%

+ Add... Capture... Delete

If you modify a tolerance after a test case has been run, then rerun the test case to apply the new tolerance value to the pass-fail results.

## Test Manager Limitations

### In this section...

- “Simulation Mode” on page 6-47
- “Callback Scripts” on page 6-47
- “Protected Models” on page 6-47
- “Parameter Overrides” on page 6-48
- “Breakpoints” on page 6-48
- “Highlight in Model” on page 6-48

### 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.

### 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** in 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...

- “Description” on page 6-50
- “Requirements” on page 6-50
- “System Under Test” on page 6-51
- “Parameter Overrides” on page 6-52
- “Callbacks” on page 6-52
- “Inputs” on page 6-53
- “Outputs” on page 6-54
- “Configuration Settings” on page 6-54
- “Simulation 1 and Simulation 2” on page 6-54
- “Equivalence Criteria” on page 6-54
- “Baseline Criteria” on page 6-55
- “Iterations” on page 6-55
- “Coverage Settings” on page 6-56


Information about the test case sections is outlined here. Double-click a test case in the **Test Browser** pane to open a tab and view all the test case sections. A baseline test case is shown as an example. For more information on which test case to use for your application, see “Introduction to the Test Manager” on page 5-2.

Baseline Test Case Enabled

[Test File](#) > [Test Suite](#) > [Baseline Test Case](#)

Baseline Test

- ▶ DESCRIPTION ?
- ▶ REQUIREMENTS ?
- ▶ SYSTEM UNDER TEST ?
- ▶ PARAMETER OVERRIDES ?
- ▶ CALLBACKS ?
- ▶ INPUTS ?
- ▶ OUTPUTS ?
- ▶ CONFIGURATION SETTINGS OVERRIDES ?
- ▶ BASELINE CRITERIA ?
- ▶ ITERATIONS ?
- ▶ COVERAGE SETTINGS ?


If a box or list in the test case shows a warning icon , then it is a required field in order for the test case to run.

## 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 license for Simulink Verification and Validation. 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”.

- 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”.


## 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 pre-load callback. See “Callbacks” on page 6-52.

---

If a new model is specified in the System Under Test section, then the model information might not be up to date. To update the model test harnesses, **Signal Builder** groups, and available configuration sets, click the **Refresh** button .

### Test Harness

If you have a test harness in your system under test, then you can select the test harness to be used for the test case. If a test harness has been added or removed from a model, then you might need to click the **Refresh** button  to view the updated list of available test harnesses.

For more information about using test harnesses, see “Refine, Test, and Debug a Subsystem” on page 2-14.


### 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 “Overriding Model Parameters in a Test Case”.

## 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.

An example of a pre-load callback script would be to add the model path:

```
addpath(C:\MATLAB\model);
```

- 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 6-47 for the limitations of callback scripts inside test cases. For information on Simulink model callbacks, see “Model Callbacks”.

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. This 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 your **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. You can use only one external input set in the **External Inputs** table to run when the test case executes. External inputs are mapped using root inport mapping. See “Identify Signal Data to Import and Map” 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 Mapping tool, see “Map Root Inport Signal Data”.

If you use Microsoft Excel files for inport mapping, then the Root Inport Mapping tool can map only double-precision scalar values from the file.

### Outputs

You can override model output settings. These settings are the same settings found in the **Data Import/Export** pane of the Model Configuration Parameters.

### Configuration Settings

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

---

**Note:** If you have selected **Override model settings** in the **Outputs** section, then these settings override the output settings in the 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 is only contained 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 all of the signal checkboxes are cleared, then nothing is compared when the test case executes.

---

For an example about how to use an equivalence test case and criteria, see “Test Two Simulations for Equivalence”.

## Baseline Criteria

This test case section is only contained in a baseline test case. You can use signal data from a MAT-file or Microsoft Excel file. Microsoft Excel files need to use the same formatting as specified by the Root Inport Mapping tool. For more information, see “Map Root Inport Signal Data”. 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.

## 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 6-30.

### **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 6-38.

# Test Models Using Inputs Generated by Simulink Design Verifier

## In this section...

“Overall Workflow” on page 6-57

“Test Case Generation Example” on page 6-57

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 Design Verifier Results Summary window appears after the analysis completes.
- 2 In the results summary window, click **Export test cases to Simulink Test**.
- 3 Select an existing test harness, or create a test harness.
- 4 Simulink Test generates the test file and test harness. In the test manager, expand the new test file in the **Test Browser** to see the individual test cases.

## Test Case Generation Example

This example shows how to generate test cases to achieve coverage objectives for a controller subsystem. It also shows how to add functional test cases from test harnesses in the model. 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.

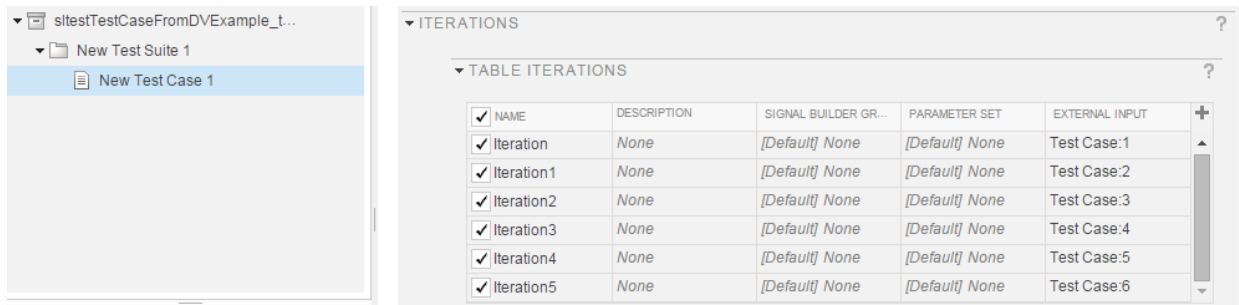
```
open_system(fullfile(docroot,'toolbox','sltest','examples',...
    'sltestTestCaseFromDVExample.slx'));
```

- 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**.

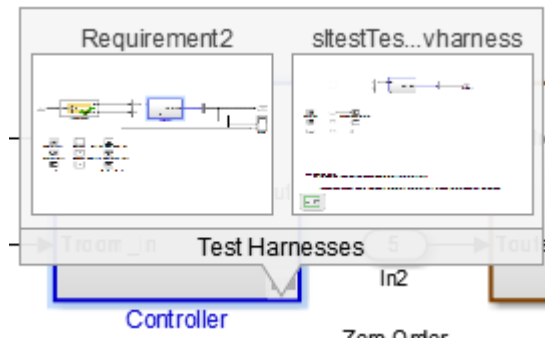
Simulink Design Verifier generates tests for the component.


- 4 In the Results Summary window, click **Export test cases to Simulink Test**.
- 5 In the Harness Selection dialog box, select New Harness. Click **OK**.

The test manager displays a new test case with several iterations.



- 6 Click the harness badge to preview the new test harness.



- 7 Add a test case to the Requirement2 test harness in the model. In the test manager, hover over the new test file name and click the **Synchronize Test File** button .
- 8 The test manager prompts you to add tests for the Requirement2 test harness. Select **Simulation** for the test type, and click **Update Test File**.

The test manager adds the Requirement2 test case to the test file.

## See Also

`s1test.import.sldvData`





# Test Manager Results and Reports

---

- “View Test Case Results” on page 7-2
- “Export Test Results and Generate Reports” on page 7-9
- “Customize Generated Reports” on page 7-13
- “Results Sections” on page 7-19

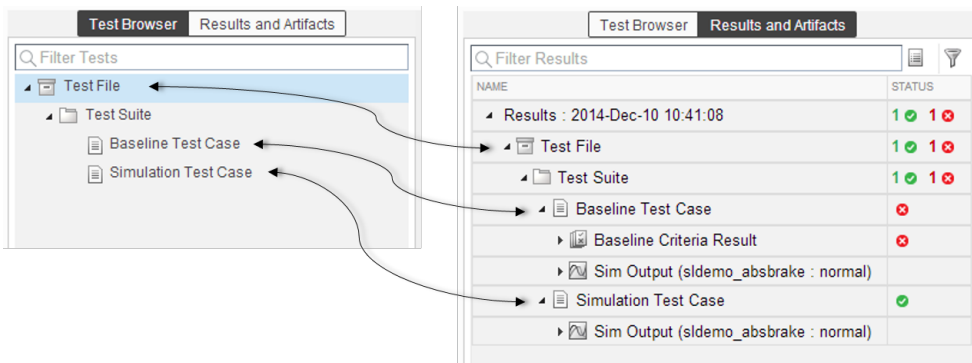
## View Test Case Results

### In this section...

“View Results Summary” on page 7-2

“Visualize Test Case Simulation Output and Criteria” on page 7-4

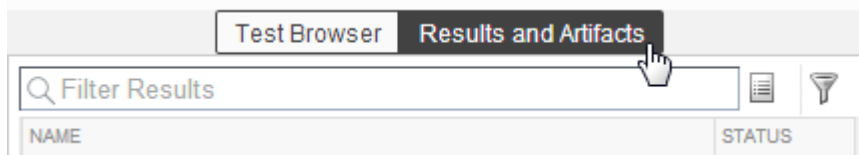
After a test case has finished running in the test manager, the test case result becomes available in the **Results and Artifacts** pane. Test results are organized in the same hierarchy as the test file, test suite, and test cases that were run from the **Test Browser** pane. In addition, the **Results 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:

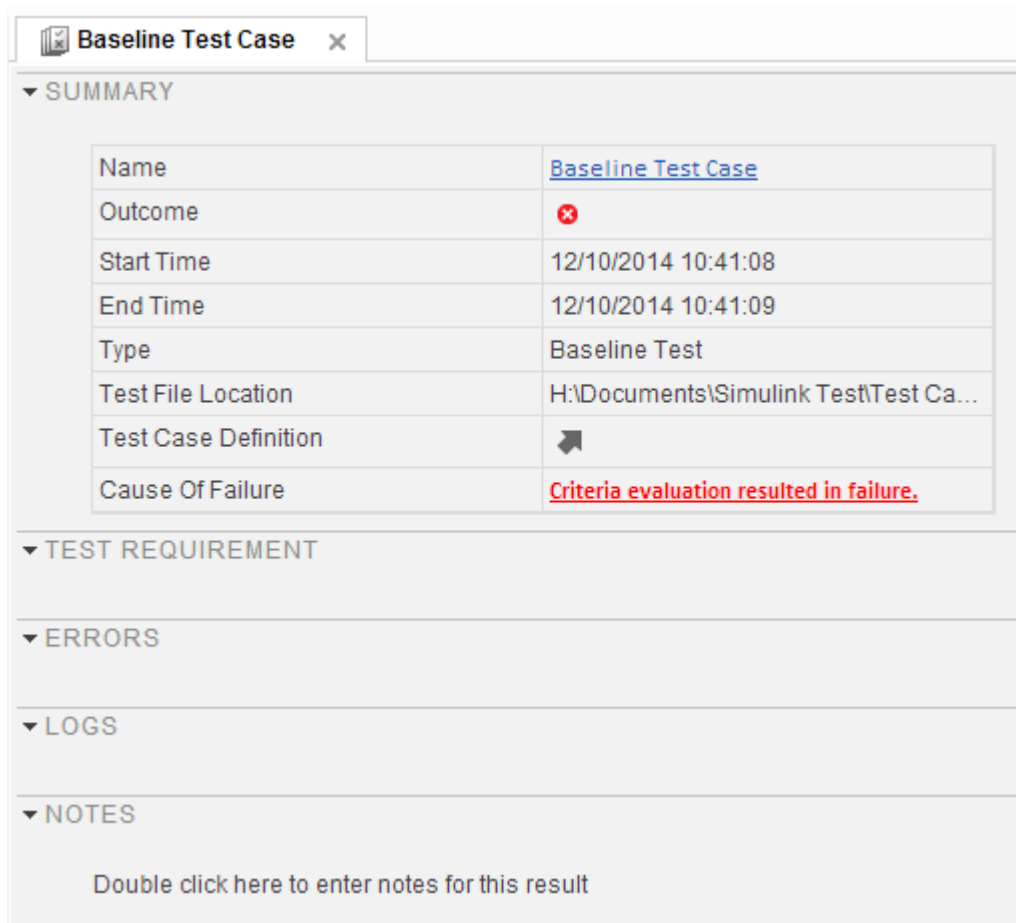
- 1 Select the **Results and Artifacts** pane.



- 2 Double-click a test case result.

NAME	STATUS
Results : 2014-Dec-10 10:41:08	1  1 
Test File	1  1 
Test Suite	1  1 
Baseline Test Case	1 
Baseline Criteria Result	1 

A tab opens containing the test case results information.



The screenshot shows a window titled "Baseline Test Case" with a close button. The window is divided into several sections:

- SUMMARY**: A table with the following data:

Name	<a href="#">Baseline Test Case</a>
Outcome	✘
Start Time	12/10/2014 10:41:08
End Time	12/10/2014 10:41:09
Type	Baseline Test
Test File Location	H:\Documents\Simulink Test\Test Ca...
Test Case Definition	➤
Cause Of Failure	<a href="#">Criteria evaluation resulted in failure.</a>
- TEST REQUIREMENT**: A section that is currently collapsed.
- ERRORS**: A section that is currently collapsed.
- LOGS**: A section that is currently collapsed.
- NOTES**: A section that is currently collapsed. Below this section is the text: "Double click here to enter notes for this result".

## 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.

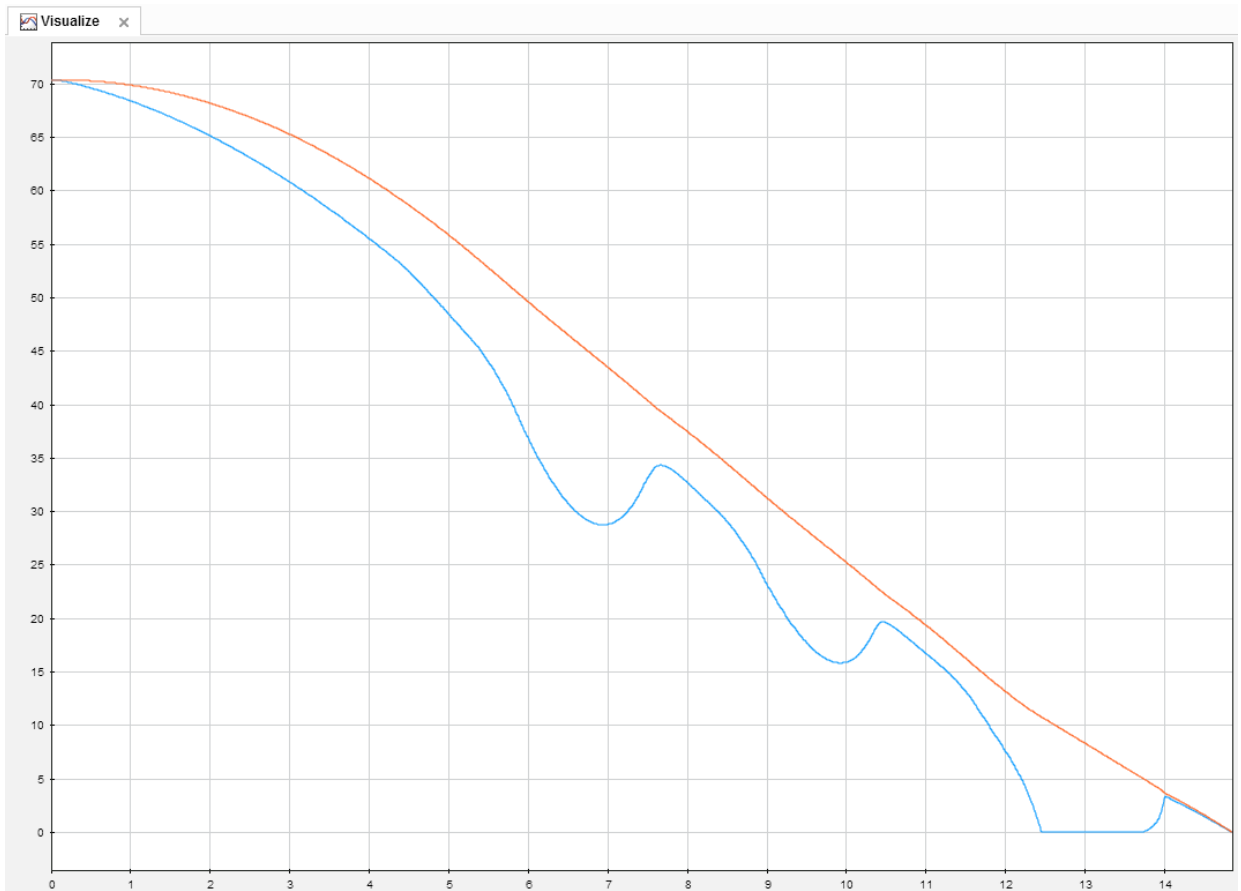
To view simulation output from a test case:

- 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		✖
<input type="radio"/> yout.Ww		✔
<input type="radio"/> yout.Vs		✖
<input type="radio"/> yout.Sd		✖
<input type="radio"/> slp		✖
Sim Output (sldemo_absbrake :		
<input checked="" type="checkbox"/> yout.Ww	—	
<input checked="" type="checkbox"/> yout.Vs	—	
<input type="checkbox"/> yout.Sd	—	
<input type="checkbox"/> slp	—	

The **Visualize** tab appears and plots the signals.

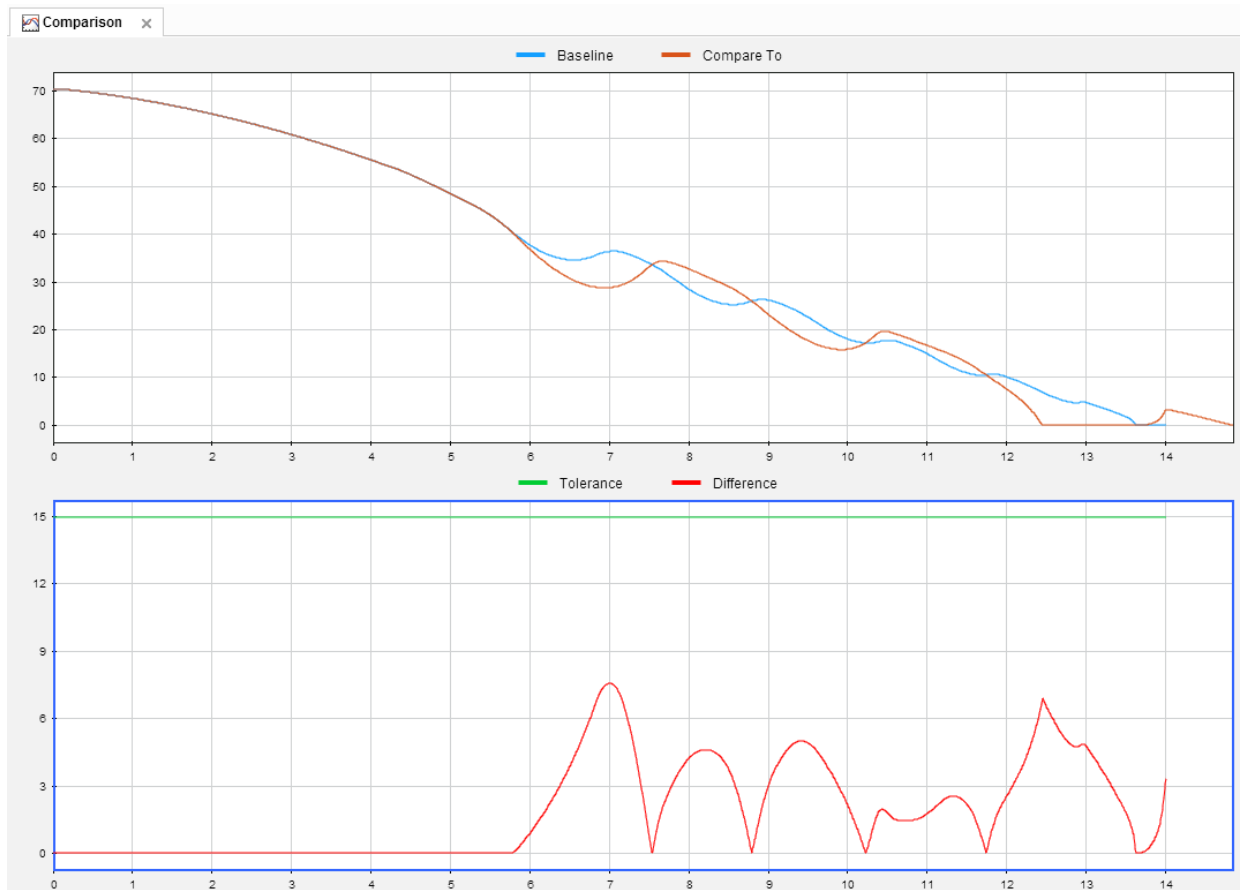


To view equivalence or baseline criteria comparisons:

- 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	✖
<input checked="" type="radio"/> yout.Ww	✔
<input type="radio"/> yout.Vs	✖
<input type="radio"/> yout.Sd	✖
<input type="radio"/> slp	✖
Sim Output (sldemo_absbrake :	
<input type="checkbox"/> yout.Ww	—
<input type="checkbox"/> yout.Vs	—
<input type="checkbox"/> yout.Sd	—
<input type="checkbox"/> 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...
“Export Results” on page 7-9
“Create a Test Results Report” on page 7-10
“Generate Report Using Microsoft Word Template” on page 7-10

Once you have run test cases and generated test results, you can export results and generate reports. Test case results are all contained in the **Results and Artifacts** pane.

## Export Results

Test results are not saved with 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 what information is included in the report, and it can be saved in three different file formats: ZIP (HTML), DOCX, and PDF.

To generate a report:

- 1 Select the **Results and Artifacts** pane.
- 2 Select results for a test file, test suite, or test case in the **Results and Artifacts** pane.

---

**Note:** You can create a report from multiple results sets, but you cannot create a report from multiple test files, test suites, or test cases within results sets.

---

- 3 From the toolstrip, click **Report**.
- 4 Choose the options of what to include in the report.
- 5 Select the **File Format** to save the report as.
- 6 Click **Create** to generate the report.

### Generate Report Using Microsoft Word Template

If you have a MATLAB Report Generator™ license, then you can create reports from a Microsoft Word template. The report can be generated to a Microsoft Word document or PDF. The report generator in Simulink Test fills in 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”.

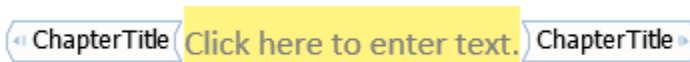
For a sample template, go to the path:

```
cd(matlabroot);  
cd('help\toolbox\sltest\examples');  
In the examples directory, open the file Template.dotx.
```

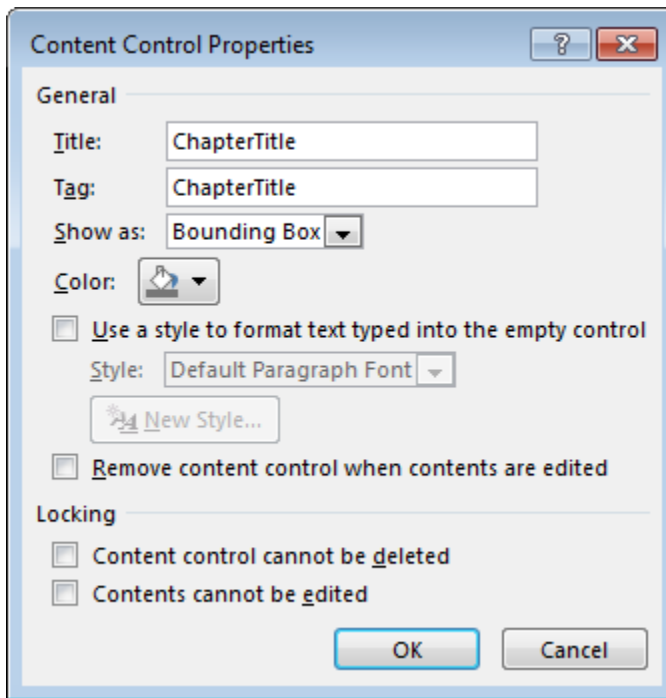
In the Microsoft Word template, you can add rich text content controls. Each Simulink Test report section can be inserted into the rich text content controls. The control names are:

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

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



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



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 toolbar, click **Report**.
- 4 Choose the options of what to include in the report.
- 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** to generate the report.

## Customize Generated Reports

### In this section...

“Inherit the Report Class” on page 7-13

“Method Hierarchy” on page 7-13

“Modify the Class” on page 7-15

“Generate a Report Using the Custom Class” on page 7-17

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. To use the custom class, you need to have a license to MATLAB Report Generator.

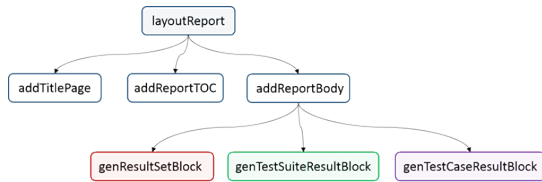
### Inherit the Report Class

In order 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 “Create Subclasses — Syntax and Techniques”. Then, add code to the inherited class or methods to create your customizations.

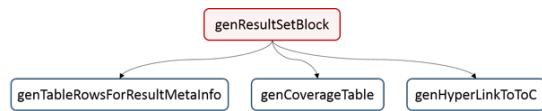
```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    %
    % Report customization code here
    %
end
```

### Method Hierarchy

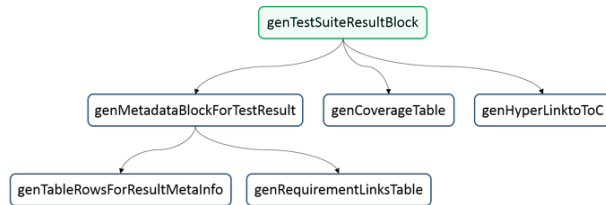
When you create the subclass, the derived class inherits methods from the `sltest.testmanager.TestResultReport` class. The body of the report is separated into three main groups: the result set block, the test suite result block, and the test case result block.



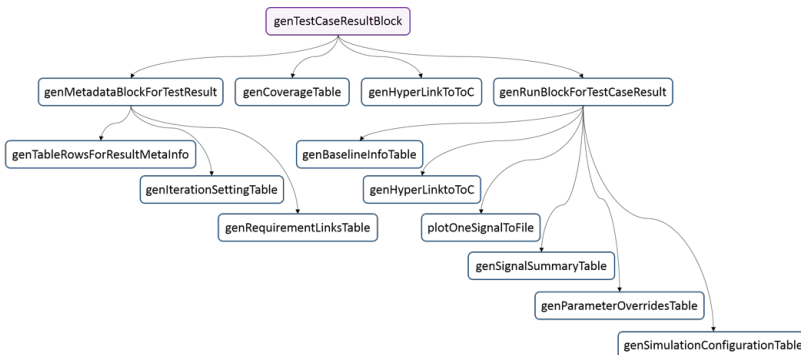
The result set block contains the result set table, the coverage table, and links to the table of contents.



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

If you want to insert your own report content or change the layout of the generated report, then you need to modify the inherited class methods. For general information about modifying methods, see “Modify Superclass Methods”.

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';
            end
        end
    end
end
```





report class in the Create Test Result Report dialog box. For the test manager to be able to use the custom report class, the class must be on the MATLAB path.

**Create Test Result Report** ? X

**Title Page Information**

Title:

Author:

Include MATLAB version

**Include in Report**

Results for:

Test requirements

Plots of criteria and assessments

Plots for simulation output and baseline

Error and log messages

Simulation metadata

Coverage results

**Output Options**

File Format:

File Name:

**Customization**

Template File:

Report Class:

**Create** **Cancel**

### See Also

`sltest.testmanager.TestResultReport` | `sltest.testmanager.report`

### Related Examples

- “Create Subclasses — Syntax and Techniques”

## Results Sections

In this section...
“Summary” on page 7-20
“Test Requirements” on page 7-20
“Iteration Settings” on page 7-21
“Errors” on page 7-21
“Logs” on page 7-21
“Description” on page 7-21
“Parameter Overrides” on page 7-21
“Coverage Results” on page 7-21

Information about test case result sections is outlined here. Double-click a test case results in the **Results and Artifacts** pane to open a results tab and view all of the test case result sections. A baseline test case result is shown as an example.

✕ **Baseline Test Case**

▼ SUMMARY

Name	<a href="#">Baseline Test Case</a>
Outcome	1 <span style="color: green;">✔</span>
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 6-50 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 6-30.

## Errors

These are simulation errors that are captured from the Simulink Diagnostic Viewer. Errors from incorrect information defined in the test case and callback scripts are also shown here.

## Logs

These are simulation warnings that are 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 6-38.



# Real-Time Testing

---

## Test Models in Real Time

### In this section...

“Overall Workflow” on page 8-2

“Real-Time Testing Considerations” on page 8-3

“Complete Basic Model Testing” on page 8-3

“Set up the Target Computer” on page 8-3

“Configure the Model or Test Harness” on page 8-4

“Add Test Cases for Real-Time Testing” on page 8-6

“Assess Real-Time Execution Using `verify` Statements” on page 8-11

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 3-40.
- 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”
- “Troubleshooting in 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.
- `slrtert.tlc` to generate system target code using Embedded Coder.

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**.

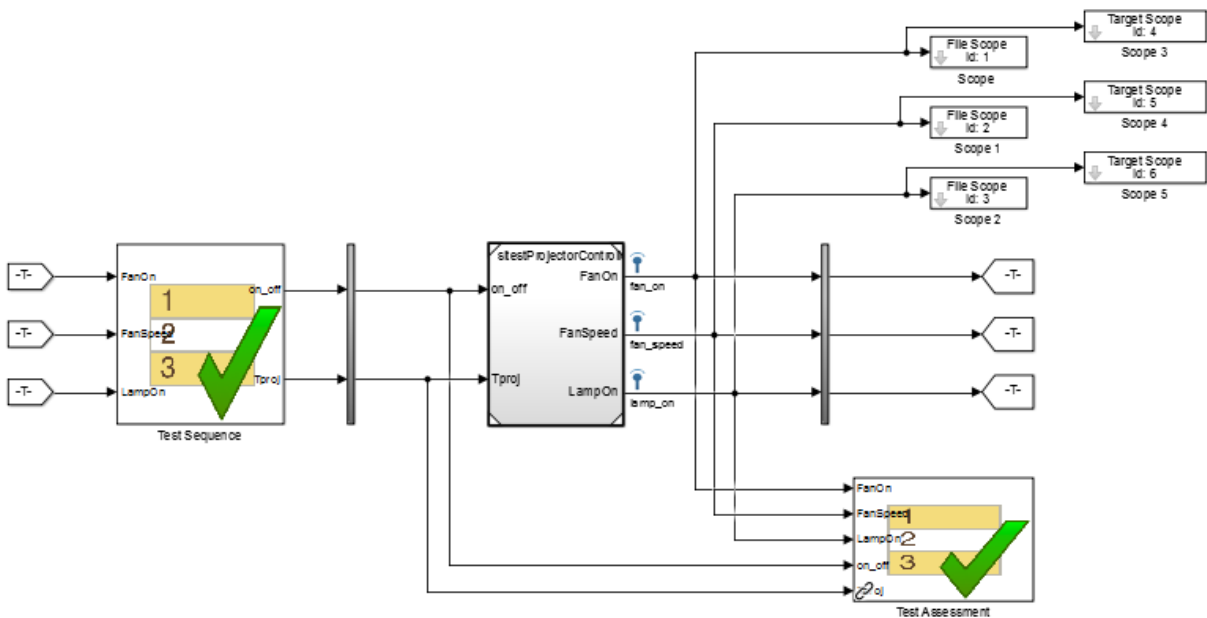
### Log Signals from Real-Time Execution

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

- Enable signal logging in the Configuration Parameters, in the Data Import/Export pane.

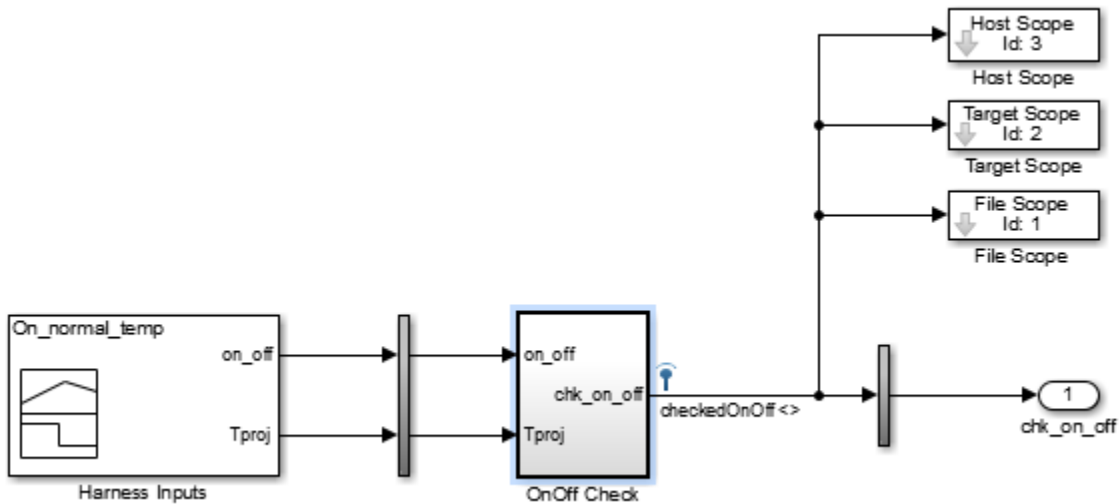
- Connect signals to **Scope** blocks from the Simulink Real-Time block library. Set the **Scope type** property to **File**.
- Name each signal of interest using the signal properties.

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.



## 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

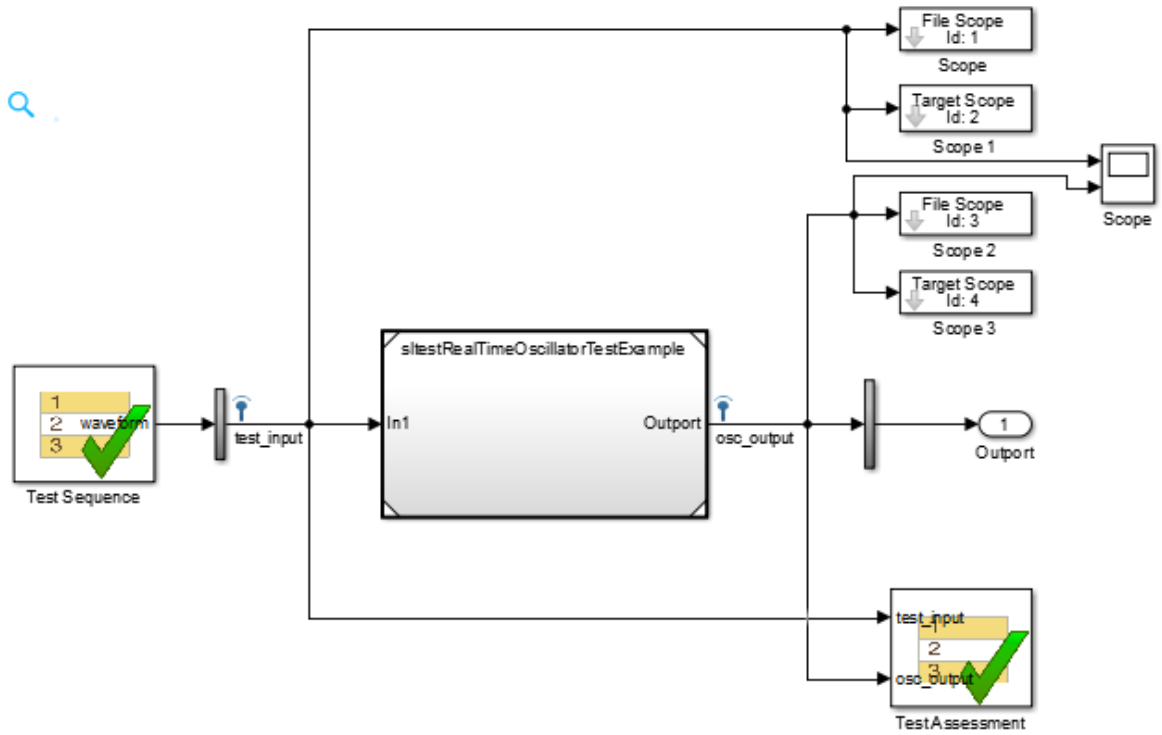
callbacks execute in a sequence along with the model load, build, download, and execute steps. Callbacks and step execution depends on how the test case loads the application.

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 6-30. 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
<b>Input</b>
<b>Output</b>
1. waveform
<b>Local</b>
<b>Constant</b>
<b>Parameter</b>
frequency

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



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;

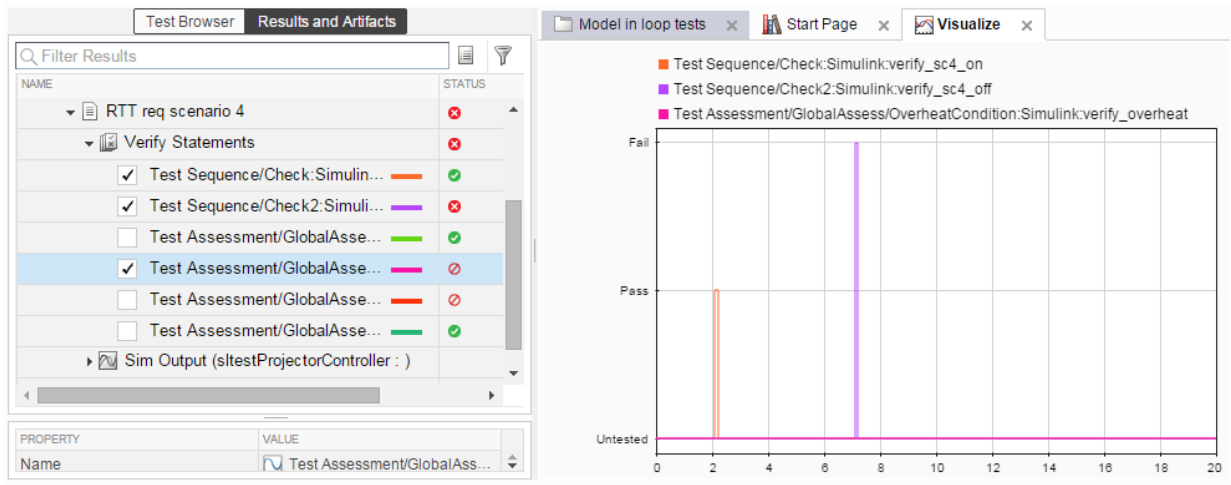
for i_iter = 1:length(freq)
    % Create iteration object
    testItr = sltestiteration();

    % 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 3-26.



## Related Examples

- “Test Real-Time Application”