

**Simulink® Test™**

User's Guide



**MATLAB® & SIMULINK®**

R2018b



# 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® Test™ User's Guide*

© COPYRIGHT 2015–2018 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)
September 2016	Online only	Revised for Version 2.1 (Release 2016b)
March 2017	Online Only	Revised for Version 2.2 (Release 2017a)
September 2017	Online Only	Revised for Version 2.3 (Release 2017b)
March 2018	Online Only	Revised for Version 2.4 (Release 2018a)
September 2018	Online Only	Revised for Version 2.5 (Release 2018b)



## Test Strategies

### 1

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

## Test Harness

### 2

<b>Test Harness and Model Relationship</b> .....	<b>2-2</b>
Test Harness Description .....	<b>2-2</b>
Harness — Model Relationship for a Model Component .....	<b>2-3</b>
Harness — Model Relationship for a Top-Level Model .....	<b>2-4</b>
Resolving Parameters .....	<b>2-5</b>
<b>Considerations and Limitations</b> .....	<b>2-7</b>
Test Harness .....	<b>2-7</b>
Test Sequence Block .....	<b>2-7</b>
<b>Test Harness Construction for Specific Model Elements</b> .....	<b>2-9</b>
Signal Conversion .....	<b>2-9</b>
Function Calls .....	<b>2-10</b>
Physical Signal Connections .....	<b>2-10</b>
Bus Signals .....	<b>2-10</b>
String Signals .....	<b>2-10</b>
Non-Graphical Connections .....	<b>2-11</b>
Export Function Models .....	<b>2-12</b>
Execution Semantics .....	<b>2-13</b>
Sample Time Specification .....	<b>2-13</b>

<b>Create Test Harnesses and Select Properties</b> .....	<b>2-15</b>
Create a Test Harness .....	2-15
Preview and Open Test Harnesses .....	2-16
Change Test Harness Properties .....	2-16
Considerations for Selecting Test Harness Properties .....	2-17
Harness Name .....	2-17
Save Test Harnesses Externally .....	2-17
Sources and Sinks .....	2-18
Add scheduler for function-calls and rates / Generate function- call signals using .....	2-18
Enable Initialize, Reset, and Terminate ports .....	2-18
Add Separate Assessment Block .....	2-19
Open Harness After Creation .....	2-19
Create without compiling the model .....	2-19
Create scalar inputs .....	2-19
Post-create callback method .....	2-19
Rebuild harness on open .....	2-20
Update Configuration Parameters and Model Workspace data on rebuild .....	2-20
Post-rebuild callback method .....	2-20
Synchronization Mode .....	2-20
Verification Modes .....	2-21
 <b>Test Harness Parameters and Signals</b> .....	 <b>2-22</b>
Test Harness Generation Without Compilation .....	2-22
Signal Conversion Subsystem .....	2-22
 <b>Refine, Test, and Debug a Subsystem</b> .....	 <b>2-24</b>
Model and Requirements .....	2-24
Create a Harness for the Controller .....	2-26
Inspect and Refine the Controller .....	2-28
Add Test Inputs and Test the Controller .....	2-28
Debug the Controller .....	2-29
 <b>Manage Test Harnesses</b> .....	 <b>2-32</b>
Internal and External Test Harnesses .....	2-32
Manage External Test Harnesses .....	2-32
Convert Between Internal and External Test Harnesses .....	2-34
Preview and Open Test Harnesses .....	2-35
Find Test Cases Associated with a Test Harness .....	2-36
Export Test Harnesses to Separate Models .....	2-36
Clone and Export a Test Harness to a Separate Model .....	2-37
Delete Test Harnesses Programmatically .....	2-39
Move and Clone Test Harnesses .....	2-41

<b>Customize Test Harnesses</b> .....	<b>2-44</b>
Callback Function Definition and Harness Information .....	<b>2-45</b>
How to Display Harness Information struct Contents .....	<b>2-47</b>
Customize a Test Harness to Create Mixed Source Types ...	<b>2-47</b>
Test Harness Callback Example .....	<b>2-49</b>
<b>Create Test Harnesses from Standalone Models</b> .....	<b>2-52</b>
Test Harness Import Workflow .....	<b>2-52</b>
Component Compatibility for Test Harness Import .....	<b>2-53</b>
.....	<b>2-54</b>
<b>Synchronize Changes Between Test Harness and Model</b> ....	<b>2-57</b>
Set Synchronization for a New Test Harness .....	<b>2-57</b>
Change Synchronization of an Existing Test Harness .....	<b>2-58</b>
Synchronize Configuration Set and Model Workspace Data ..	<b>2-58</b>
Check for Unsynchronized Component Differences .....	<b>2-59</b>
Rebuild a Test Harness .....	<b>2-59</b>
Push Changes from Test Harness to Model .....	<b>2-60</b>
Check Component and Push Parameter to Main Model .....	<b>2-60</b>
<b>Test Library Blocks</b> .....	<b>2-64</b>
Library Testing Workflow .....	<b>2-64</b>
Library and Linked Subsystem Test Harnesses .....	<b>2-64</b>
Edit Library Block from a Test Harness .....	<b>2-66</b>
Testing a Library and a Linked Block .....	<b>2-66</b>

## Test Sequences and Assessments

# 3

<b>Test Sequence Basics</b> .....	<b>3-2</b>
Structure of a Test Sequence .....	<b>3-2</b>
Test Sequence Hierarchy .....	<b>3-2</b>
Transition Types .....	<b>3-2</b>
Create a Basic Test Sequence .....	<b>3-4</b>
Create Basic Test Assessments .....	<b>3-5</b>
<b>Assessment Basics</b> .....	<b>3-10</b>
Overview .....	<b>3-10</b>
Comparing Simulation to Baseline Data or Another Simulation .....	<b>3-11</b>

Post-Processing Simulation Output .....	3-11
Run-Time Assessments .....	3-12
<b>Activate verify Statements .....</b>	<b>3-14</b>
Verify Based on the Active Test Step .....	3-14
Verify Using Signal Conditions .....	3-17
Verify Multiple Conditions at a Time .....	3-18
<b>Test Sequence Editor .....</b>	<b>3-20</b>
Input, Output, and Data Management .....	3-20
Add and Delete Test Steps .....	3-21
Copy Test Steps .....	3-22
Reorder Test Steps and Transitions .....	3-22
Change Test Step Hierarchy .....	3-22
Standard Test Step Transitions .....	3-23
When Decomposition Transitions .....	3-24
<b>Test Step Actions and Transitions .....</b>	<b>3-32</b>
Transition Between Steps Using Temporal or Signal Conditions .....	3-32
Temporal Operators .....	3-33
Transition Operators .....	3-34
Use Messages in Test Sequences .....	3-35
<b>Signal Generation Functions .....</b>	<b>3-41</b>
Sinusoidal and Random Number Functions in Test Sequences .....	3-41
Using an External Function from a Test Sequence Block .....	3-43
Signal Generation Functions .....	3-45
<b>Run-Time Assessments .....</b>	<b>3-49</b>
verify .....	3-49
assert .....	3-51
Assessment Statements .....	3-52
Logical Operators .....	3-53
Relational Operators .....	3-54
<b>Programmatically Create a Test Sequence .....</b>	<b>3-56</b>
<b>Syntax for Test Sequences and Assessments .....</b>	<b>3-60</b>
Assessment Statements .....	3-60
Temporal Operators .....	3-62
Transition Operators .....	3-63



Signal Generation Functions .....	3-64
Logical Operators .....	3-67
Relational Operators .....	3-68
<b>Debug a Test Sequence .....</b>	<b>3-69</b>
View Test Step Execution During Simulation .....	3-69
Set Breakpoints to Enable Debugging .....	3-69
View Data Values During Simulation .....	3-70
Step Through Simulation .....	3-71
<b>Test Downshift Points of a Transmission Controller .....</b>	<b>3-72</b>
<b>Test Assessment Reuse .....</b>	<b>3-79</b>
Reuse Test Assessments Using a Library .....	3-79
<b>View Graphical Results From Model Verification Library ...</b>	<b>3-84</b>

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

# 4

<b>SIL Verification for a Subsystem .....</b>	<b>4-2</b>
Create a SIL Verification Harness for a Controller .....	4-2
Configure and Simulate a SIL Verification Harness .....	4-4
Compare the SIL Block and Model Controller Outputs .....	4-5
<b>Test Integrated Code .....</b>	<b>4-7</b>
Test Integrated C Code .....	4-7
Test Code in S-Functions .....	4-8
S-Function Testing Example .....	4-8

## Simulink Test Manager Introduction

# 5

<b>Introduction to Test Manager .....</b>	<b>5-2</b>
Start Test Manager .....	5-2
Create Tests and Understand the Test Hierarchy .....	5-2
View Test Results .....	5-4

Share Results . . . . .	5-4
Compare Test Files . . . . .	5-4

## Test Manager Test Cases

# 6

<b>Manage Test File Dependencies . . . . .</b>	<b>6-2</b>
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>	<b>6-9</b>
Create the Test Case . . . . .	6-9
Run the Test Case and View Results . . . . .	6-10
<b>Test a Simulation for Run-Time Errors . . . . .</b>	<b>6-13</b>
Configure the Model . . . . .	6-13
Create the Test Case . . . . .	6-14
Run the Test Case . . . . .	6-14
View Test Results . . . . .	6-15
<b>Automatically Create Test Cases for Your Model . . . . .</b>	<b>6-16</b>
Creating Test Cases from Model Elements . . . . .	6-16
Generating Test Cases from a Model . . . . .	6-16
<b>Generate Tests for a Subsystem . . . . .</b>	<b>6-22</b>
Generate the Subsystem Test Case . . . . .	6-22
<b>Synchronize Test Cases to Model Changes . . . . .</b>	<b>6-24</b>
<b>Specify Microsoft Excel File Format for Signal Data . . . . .</b>	<b>6-25</b>
Basic Excel File Format . . . . .	6-25
Input and Output Data . . . . .	6-26
Parameters . . . . .	6-26
Signal Tolerances . . . . .	6-27
Simulation for Equivalence Tests . . . . .	6-28
Scalar, Multidimensional, Complex, and Bus Signals . . . . .	6-28
Data Type, Unit, Interpolation, and Block Path/Port Index . . . . .	6-30
Function-Call Execution Times . . . . .	6-33
Limitations . . . . .	6-33

<b>Author a Test in Excel</b> .....	<b>6-34</b>
Generate an Excel Template .....	<b>6-34</b>
<b>Create Test with Data from Excel</b> .....	<b>6-39</b>
Create a Test Case from an Excel Spreadsheet .....	<b>6-39</b>
Import an Excel Spreadsheet into a Test Case .....	<b>6-40</b>
<b>Use External Inputs in Test Cases</b> .....	<b>6-42</b>
Mapping Status .....	<b>6-42</b>
Add a MAT-File as an External Input .....	<b>6-43</b>
Add Microsoft Excel File as Input .....	<b>6-43</b>
Importing Microsoft® Excel® Data .....	<b>6-44</b>
<b>Create Data Files to Use as Test Inputs</b> .....	<b>6-49</b>
Create a Microsoft Excel File for Input Data .....	<b>6-49</b>
Create a MAT-File for Input Data .....	<b>6-50</b>
<b>Run Tests in Multiple Releases</b> .....	<b>6-52</b>
Considerations for Testing in Multiple Releases .....	<b>6-52</b>
Add Releases Using Test Manager Preferences .....	<b>6-53</b>
Run Baseline Tests in Multiple Releases .....	<b>6-53</b>
Run Equivalence Tests in Multiple Releases .....	<b>6-54</b>
Run Simulation Tests in Multiple Releases .....	<b>6-55</b>
<b>Examine Test Failures and Modify Baselines</b> .....	<b>6-57</b>
Examine Test Failure Signals and Update Baseline Test .....	<b>6-57</b>
Manually Update Signal Data in a Baseline .....	<b>6-60</b>
<b>Automate Tests Programmatically</b> .....	<b>6-63</b>
List of Functions and Classes .....	<b>6-63</b>
Create and Run a Baseline Test Case .....	<b>6-65</b>
Create and Run an Equivalence Test Case .....	<b>6-66</b>
Run a Test Case and Collect Coverage .....	<b>6-67</b>
Create and Run Test Case Iterations .....	<b>6-68</b>
<b>Run Combinations of Tests Using Iterations</b> .....	<b>6-70</b>
Create Table Iterations .....	<b>6-70</b>
Create Scripted Iterations .....	<b>6-73</b>
Capture Baseline Data from Iterations .....	<b>6-76</b>
Sweep Through a Set of Parameters .....	<b>6-79</b>
<b>Collect Coverage in Tests</b> .....	<b>6-81</b>
Enable and Collect Coverage for a Test File .....	<b>6-81</b>

Considerations for Collecting Coverage in Test Harnesses . . .	<b>6-84</b>
<b>Run Tests Using Parallel Execution</b> . . . . .	<b>6-86</b>
Use Parallel Execution . . . . .	<b>6-86</b>
When Do Tests Benefit from Using Parallel Execution? . . . . .	<b>6-87</b>
<b>Apply Tolerances to Test Criteria</b> . . . . .	<b>6-88</b>
Modify Criteria Tolerances . . . . .	<b>6-88</b>
Change Leading Tolerance in a Baseline Comparison Test . . . . .	<b>6-88</b>
<b>Test Manager Limitations</b> . . . . .	<b>6-93</b>
Simulation Mode . . . . .	<b>6-93</b>
Callback Scripts . . . . .	<b>6-93</b>
Protected Models . . . . .	<b>6-93</b>
Parameter Overrides . . . . .	<b>6-94</b>
Breakpoints . . . . .	<b>6-94</b>
Highlight in Model . . . . .	<b>6-94</b>
<b>Test Sections</b> . . . . .	<b>6-95</b>
Select Releases for Testing . . . . .	<b>6-95</b>
Set Preferences to Display Test Sections . . . . .	<b>6-96</b>
Select releases for simulation . . . . .	<b>6-96</b>
Tags . . . . .	<b>6-96</b>
Description . . . . .	<b>6-96</b>
Requirements . . . . .	<b>6-97</b>
System Under Test . . . . .	<b>6-97</b>
Parameter Overrides . . . . .	<b>6-98</b>
Callbacks . . . . .	<b>6-98</b>
Inputs . . . . .	<b>6-100</b>
Simulation Outputs . . . . .	<b>6-100</b>
Configuration Setting Overrides . . . . .	<b>6-101</b>
Simulation 1 and Simulation 2 . . . . .	<b>6-102</b>
Equivalence Criteria . . . . .	<b>6-102</b>
Baseline Criteria . . . . .	<b>6-102</b>
Custom Criteria . . . . .	<b>6-104</b>
Iterations . . . . .	<b>6-104</b>
Coverage Settings . . . . .	<b>6-105</b>
Test File Options . . . . .	<b>6-105</b>
<b>Test Models Using Inputs Generated by Simulink Design Verifier</b> . . . . .	<b>6-106</b>
Overall Workflow . . . . .	<b>6-106</b>
Test Case Generation Example . . . . .	<b>6-107</b>

<b>Apply Custom Criteria to Test Cases</b> .....	<b>6-111</b>
MATLAB Testing Framework .....	<b>6-111</b>
Define a Custom Criteria Script .....	<b>6-112</b>
Reuse Custom Criteria and Debug Using Breakpoints .....	<b>6-113</b>
Assess the Damping Ratio of a Flutter Suppression System .....	<b>6-115</b>
Custom Criteria Programmatic Interface Example .....	<b>6-120</b>
<b>Create, Store, and Open MATLAB Figures</b> .....	<b>6-122</b>
Create a Custom Figure for a Test Case .....	<b>6-122</b>
Include Figures in a Report .....	<b>6-124</b>
<b>Test Models Using MATLAB Unit Test</b> .....	<b>6-125</b>
Overall Workflow .....	<b>6-125</b>
Considerations .....	<b>6-125</b>
Comparison of Test Nomenclature .....	<b>6-126</b>
Basic Workflow Using MATLAB® Unit Test .....	<b>6-127</b>
<b>Tests for Continuous Integration</b> .....	<b>6-129</b>
Test a Model for Continuous Integration Systems .....	<b>6-129</b>
Model Coverage Results for Continuous Integration .....	<b>6-132</b>
<b>Filter Test Execution and Results</b> .....	<b>6-134</b>
Add Tags .....	<b>6-134</b>
Filter Tests and Results .....	<b>6-134</b>
Run Filtered Tests .....	<b>6-134</b>

## Test Manager Results and Reports

# 7

<b>View Test Case Results</b> .....	<b>7-2</b>
View Results Summary .....	<b>7-2</b>
Visualize Test Case Simulation Output and Criteria .....	<b>7-4</b>
<b>Export Test Results and Generate Reports</b> .....	<b>7-9</b>
Export Results .....	<b>7-9</b>
Create a Test Results Report .....	<b>7-10</b>
Save Reporting Options with a Test File .....	<b>7-10</b>
Generate Reports Using Templates .....	<b>7-11</b>
Generating a Test Results Report .....	<b>7-13</b>

<b>Customize Test Reports</b> .....	<b>7-15</b>
Inherit the Report Class .....	<b>7-15</b>
Method Hierarchy .....	<b>7-15</b>
Modify the Class .....	<b>7-17</b>
Generate a Report Using the Custom Class .....	<b>7-19</b>
 <b>Append Code to a Test Report</b> .....	 <b>7-21</b>
 <b>Results Sections</b> .....	 <b>7-24</b>
Summary .....	<b>7-25</b>
Test Requirements .....	<b>7-25</b>
Iteration Settings .....	<b>7-26</b>
Errors .....	<b>7-26</b>
Logs .....	<b>7-26</b>
Description .....	<b>7-26</b>
Parameter Overrides .....	<b>7-26</b>
Coverage Results .....	<b>7-26</b>

## **Real-Time Testing**

# 8

<b>Test Models in Real Time</b> .....	<b>8-2</b>
Overall Workflow .....	<b>8-2</b>
Real-Time Testing Considerations .....	<b>8-3</b>
Complete Basic Model Testing .....	<b>8-3</b>
Set up the Target Computer .....	<b>8-3</b>
Configure the Model or Test Harness .....	<b>8-4</b>
Add Test Cases for Real-Time Testing .....	<b>8-6</b>
Assess Real-Time Execution Using verify Statements .....	<b>8-11</b>
 <b>Reuse Desktop Test Cases for Real-Time Testing</b> .....	 <b>8-13</b>
Convert Desktop Test Cases to Real-Time .....	<b>8-13</b>
Use External Data for Real-Time Tests .....	<b>8-13</b>
Example .....	<b>8-14</b>

<b>Test Model Against Requirements and Report Results</b> . . . . .	<b>9-2</b>
Requirements - Test Traceability Overview . . . . .	<b>9-2</b>
Display the Requirements and Test Case . . . . .	<b>9-3</b>
Link Requirements to Tests . . . . .	<b>9-4</b>
Run the Test . . . . .	<b>9-5</b>
Report the Results . . . . .	<b>9-6</b>
<b>Analyze a Model for Standards Compliance and Design</b>	
<b>Errors</b> . . . . .	<b>9-8</b>
Standards and Analysis Overview . . . . .	<b>9-8</b>
Check Model for Style Guideline Violations and Design	
Errors . . . . .	<b>9-8</b>
<b>Perform Functional Testing and Analyze Test Coverage</b> . . . . .	<b>9-11</b>
Functional Testing and Coverage Analysis Overview . . . . .	<b>9-11</b>
Incrementally Increase Test Coverage Using Test Case	
Generation . . . . .	<b>9-11</b>
<b>Analyze Code and Test Software-in-the-Loop</b> . . . . .	<b>9-15</b>
Code Analysis and Testing Software-in-the-Loop Overview . .	<b>9-15</b>
Analyze Code for Defects, Metrics, and MISRA C:2012 . . . . .	<b>9-15</b>





# Test Strategies

---

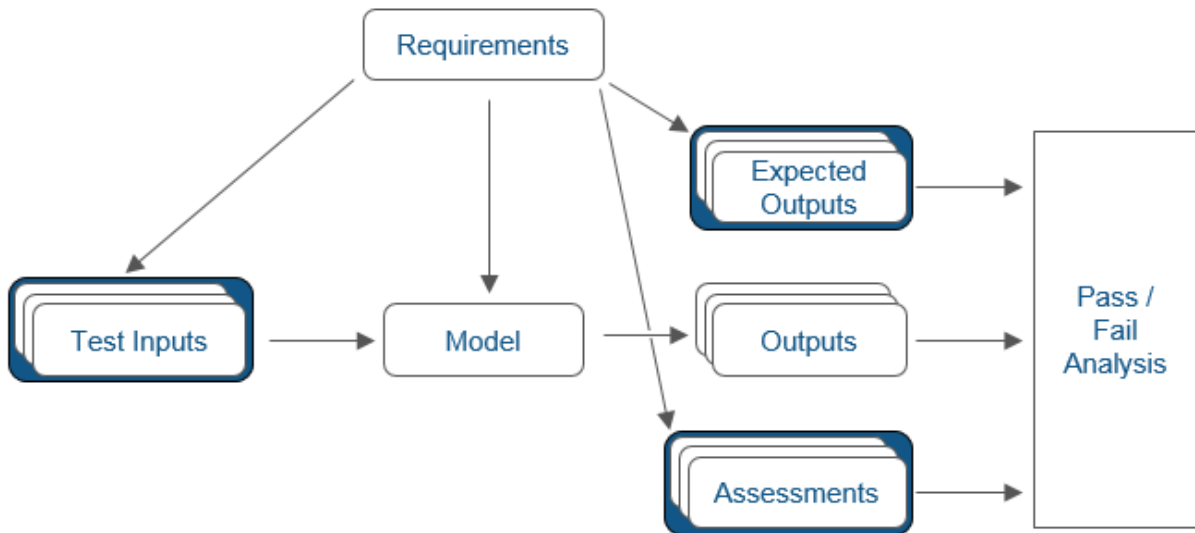
## Link Tests to Requirements

**In this section...**

“Requirements Traceability Considerations” on page 1-2

“Establish Requirements Traceability for Testing” on page 1-3

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



### Requirements Traceability Considerations

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

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

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

## Establish Requirements Traceability for Testing

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


### Requirements Traceability for Test Harnesses

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

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

### Requirements Traceability for Test Sequences

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

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

## Requirements Traceability for Test Cases

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

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

## Requirements Traceability Example

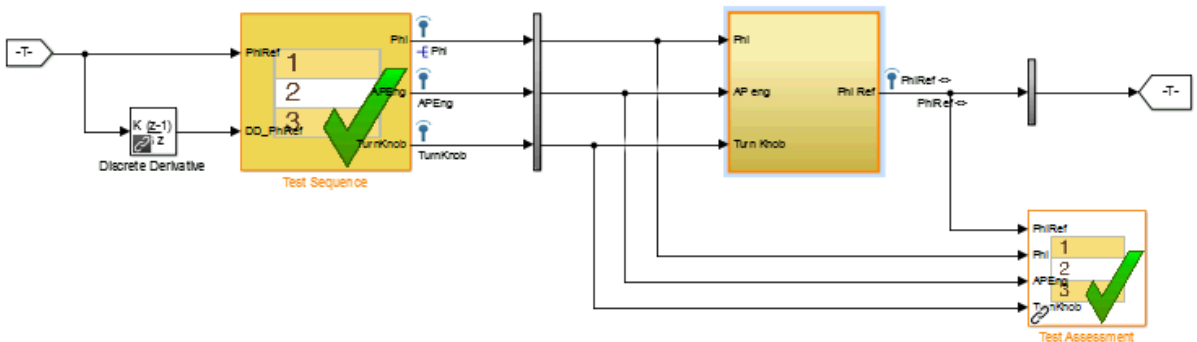
This example demonstrates adding requirements links to a test harness and test sequence. The model is a component of an autopilot roll control system. This example requires Simulink Test and Simulink Requirements.

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

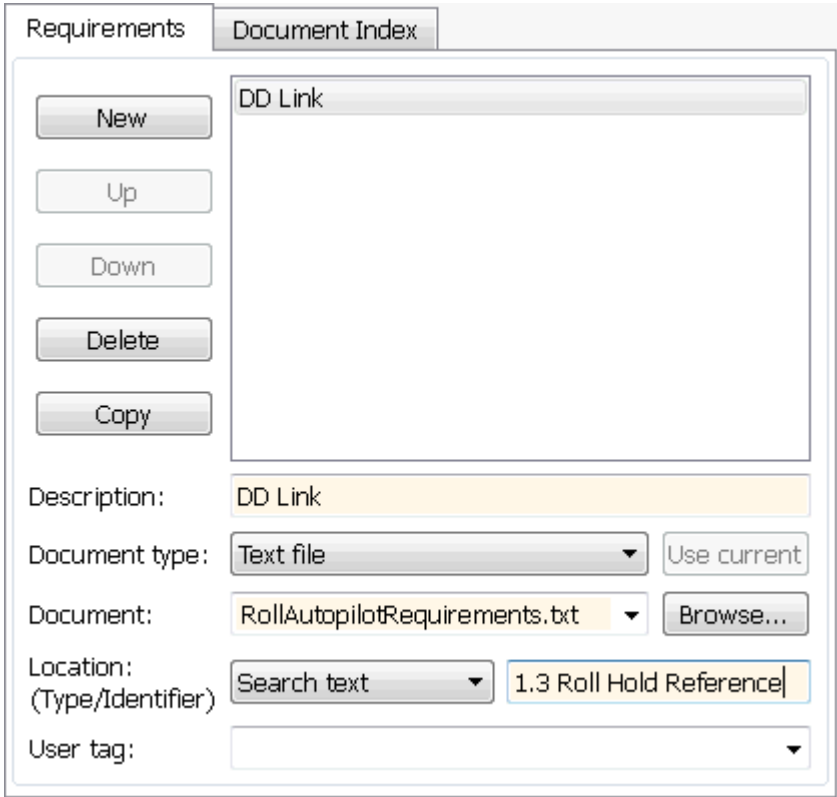
```
open AutopilotTestFile.mldatx,
open_system RollAutopilotMdlRef,
sltest.harness.open('RollAutopilotMdlRef/Roll Reference',...
'RollReference_Requirement1_3')
```

- 2 In the test harness, select **Analysis > Requirements > Highlight Model**.

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



- 3 Add traceability to the Discrete Derivative block.
  - a Right-click the Discrete Derivative block and select **Requirements > Open Outgoing Links dialog**.
  - b In the **Requirements** tab, click **New**.
  - c Enter the following to establish the link:
    - Description: DD link
    - Document type: Text file
    - Document: RollAutopilotRequirements.txt
    - Location: 1.3 Roll Hold Reference



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

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

## 1.3 Roll Hold Reference

Navigate to test harness using MATLAB command:

```
web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu
```

### REQUIREMENT

1.3.1 When roll hold mode becomes the active mode the roll hold

Navigate to test step using MATLAB command:

```
web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu
```

1.3.1.1. The roll hold reference shall be set to zero if the act

Navigate to test step using MATLAB command:

```
web('http://localhost:31415/matlab/feval/rmiobjnavigate?argu
```

- 5 Open the Test Sequence block. Add a requirements link that links the InitializeTest step to the test case.
  - a In the Test Manager, highlight Requirement 1.3 Test in the test browser.
  - b Right-click the InitializeTest step in the Test Sequence Editor. Select **Requirements > Link to Current Test Case**.

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

Step	Transition
<pre>InitializeTest Phi = 0; APEng = false; TurnKnob = 0; % Initializes test sequence outputs</pre>	<pre>1. true</pre>

## **See Also**

### **Related Examples**

- “Requirements-Based Testing for Model Development”





# Test Harness

---

- “Test Harness and Model Relationship” on page 2-2
- “Considerations and Limitations” on page 2-7
- “Test Harness Construction for Specific Model Elements” on page 2-9
- “Create Test Harnesses and Select Properties” on page 2-15
- “Test Harness Parameters and Signals” on page 2-22
- “Refine, Test, and Debug a Subsystem” on page 2-24
- “Manage Test Harnesses” on page 2-32
- “Customize Test Harnesses” on page 2-44
- “Create Test Harnesses from Standalone Models” on page 2-52
- “Synchronize Changes Between Test Harness and Model” on page 2-57
- “Test Library Blocks” on page 2-64

## Test Harness and Model Relationship

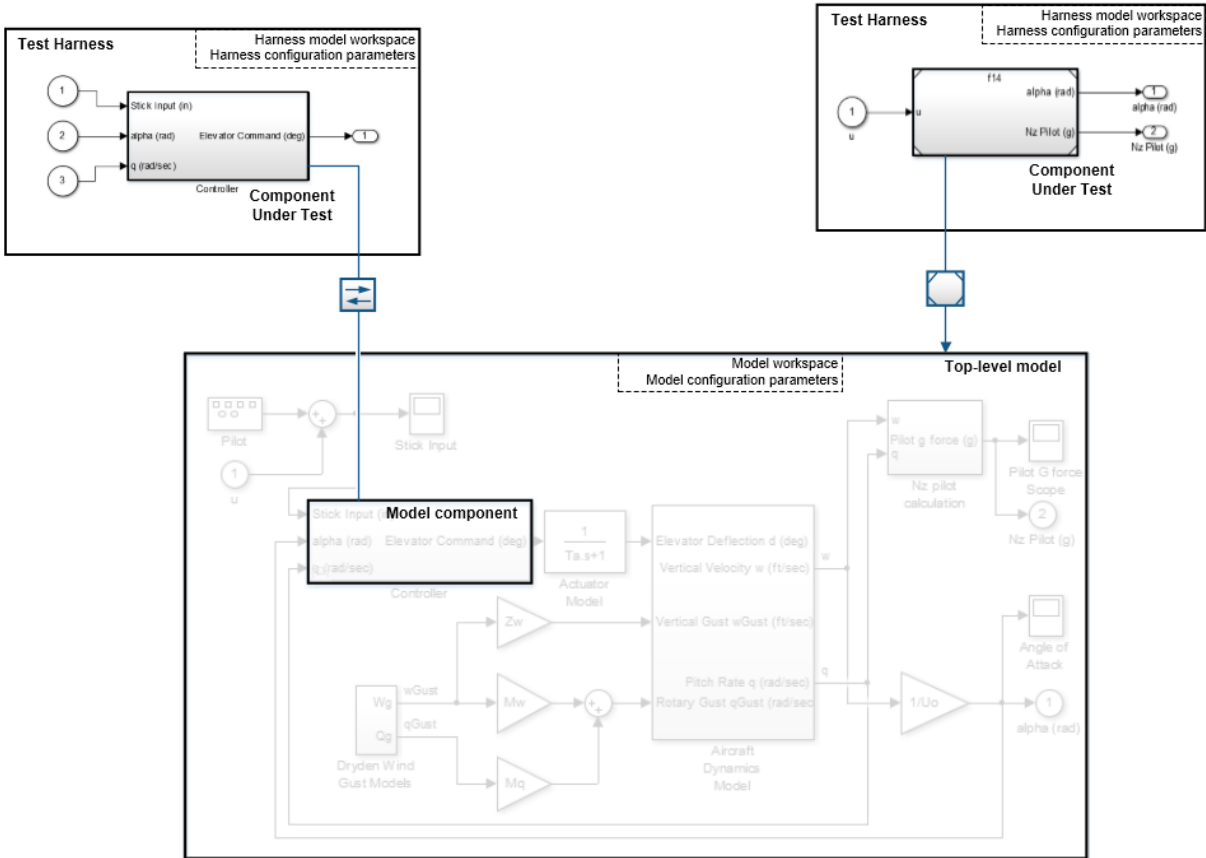
In this section...
“Test Harness Description” on page 2-2
“Harness — Model Relationship for a Model Component” on page 2-3
“Harness — Model Relationship for a Top-Level Model” on page 2-4
“Resolving Parameters” on page 2-5

### 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.

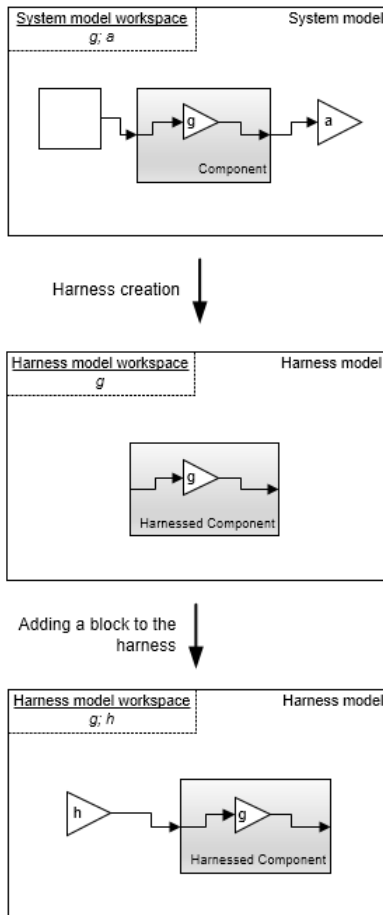


## Harness – Model Relationship for a Model Component

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

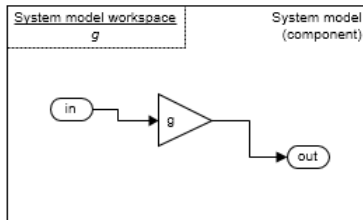
This example shows a test harness for a component that contains a Gain block. The harness model workspace contains a copy of the parameter  $g$  because  $g$  defines a part of the component.

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.

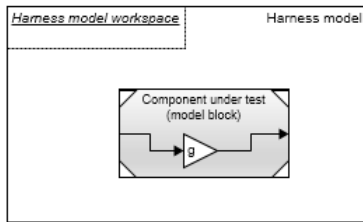


### Harness — Model Relationship for a Top-Level Model

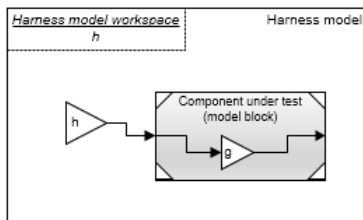
When you associate a harness with the top level of the main model, the harness model workspace does not contain copies of parameters relevant to the component. The component under test is a Model block referencing the main model, and parameters remain in the main model workspace. In this example, the 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.



Harness creation  
↓



Adding a block to the harness  
↓



## 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.

## **See Also**

### **More About**

- “Componentization Guidelines” (Simulink)

## Considerations and Limitations

In this section...
“Test Harness” on page 2-7
“Test Sequence Block” on page 2-7

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

### Test Sequence Block

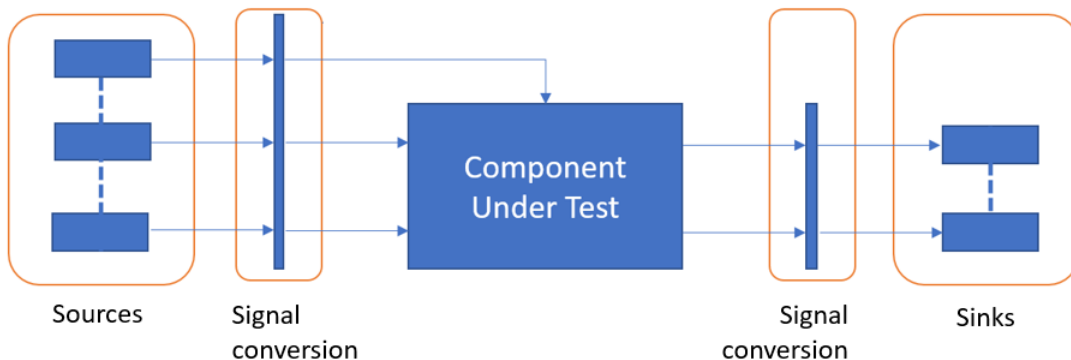
- HDL code generation is not supported for the Test Sequence block.
- The Test Sequence Editor changes the following syntax automatically:
  - Duplicate test step names. For example, if `step_1` exists, and you change another step name to `step_1`, the step name you change automatically changes to `step_2`.
  - 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)`.



## Test Harness Construction for Specific Model Elements

A test harness consists of one or more source blocks that drive the component under test, which drives one or more sink blocks. Test harness construction configures signal attributes, function calls, data stores, and execution semantics. When possible, the test harness matches signal attributes at the sources, sinks, and component interface. For more information on selecting sources and sinks, see “Sources and Sinks” on page 2-18.



### Signal Conversion

Signal conversion subsystems adapt the signal interface of the source and sink blocks to the graphical interface of the component. The graphical interface of the component includes input signals, output signals, and action, trigger, or enable inputs. The test harness compiles the main model to determine signal attributes: Data type, dimensions, and complexity of each signal. Signal attributes are adapted to the sources during harness construction in one of two ways:

- 1 Source blocks that can generate signals with the compiled attributes are configured to do so.
- 2 If a source block cannot generate signals with the compiled attributes, signal attribute blocks in the signal conversion subsystem adapt the output of the source blocks. Signal attribute blocks include Reshape, Rate Transition and Data Type Conversion blocks.

## Function Calls

### Function Call Drivers

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

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

### Function Call Outputs

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

## Physical Signal Connections

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

## Bus Signals

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

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

## String Signals

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

## String Inputs

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

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

## String Outputs

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

## Non-Graphical Connections

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

### Goto-From Connections

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

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

### **Data Store Memory**

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

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

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

### **Simulink Function Definitions**

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

### **Export Function Models**

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

The scheduler Test Sequence block includes a test step containing:

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

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

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

## Execution Semantics

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

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

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

## Sample Time Specification

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

- Source blocks that support the desired rate are configured to do so, and the signal conversion subsystem contains a Signal Specification block with the rate specification.
- Test harness construction does not configure source blocks that cannot support the desired rate.
  - If the desired rate is periodic discrete or fixed-in-minor-step, the test harness contains a Rate Transition block in the signal conversion subsystem.
  - If the desired rate is continuous, the execution semantics are determined by the solver. The signal conversion subsystem does not contain a Rate Transition block.

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

### **See Also**

“Create Test Harnesses and Select Properties” on page 2-15

## Create Test Harnesses and Select Properties

### In this section...

“Create a Test Harness” on page 2-15

“Preview and Open Test Harnesses” on page 2-16

“Change Test Harness Properties” on page 2-16

“Considerations for Selecting Test Harness Properties” on page 2-17

“Harness Name” on page 2-17

“Save Test Harnesses Externally” on page 2-17

“Sources and Sinks” on page 2-18

“Add scheduler for function-calls and rates / Generate function-call signals using” on page 2-18

“Enable Initialize, Reset, and Terminate ports” on page 2-18

“Add Separate Assessment Block” on page 2-19

“Open Harness After Creation” on page 2-19

“Create without compiling the model” on page 2-19

“Create scalar inputs” on page 2-19

“Post-create callback method” on page 2-19

“Rebuild harness on open” on page 2-20

“Update Configuration Parameters and Model Workspace data on rebuild” on page 2-20

“Post-rebuild callback method” on page 2-20

“Synchronization Mode” on page 2-20

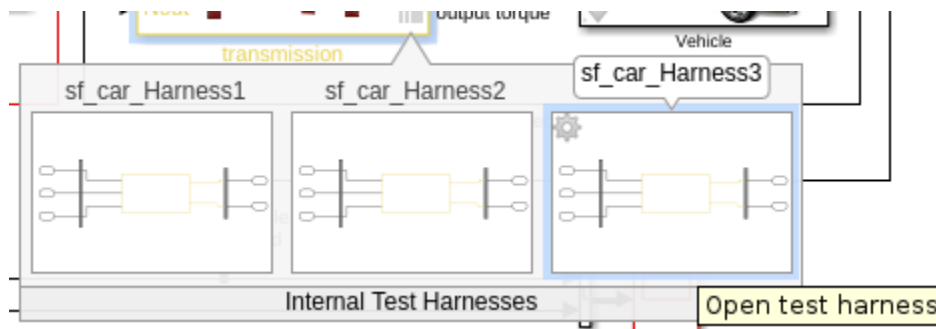
“Verification Modes” on page 2-21

### Create a Test Harness

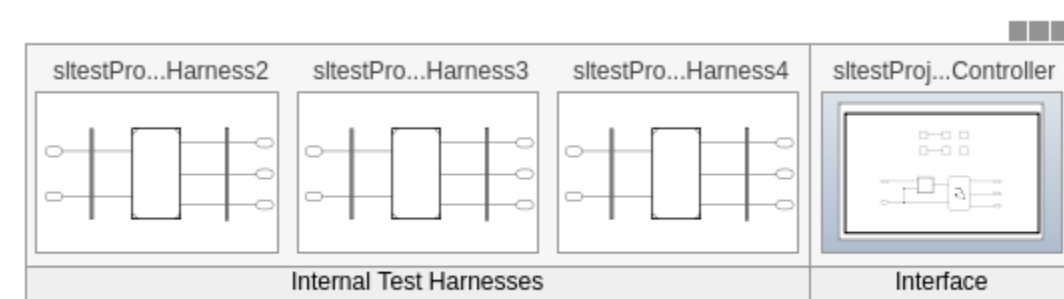
To create a test harness for a top-level model, select **Analysis > Test Harness > Create for Model**. To create a test harness for a subsystem, select the subsystem and select **Analysis > Test Harness > Create for <subsystem name>**. Set test harness properties using the Create Test Harness dialog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

## Preview and Open Test Harnesses


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



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

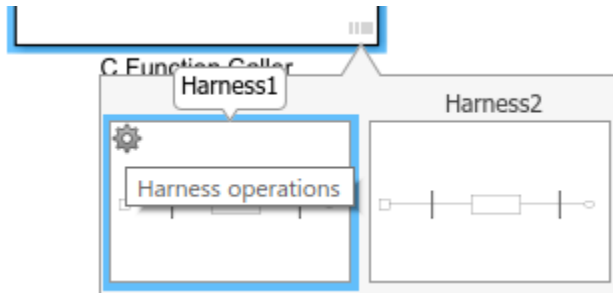


## Change Test Harness Properties

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

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





## Considerations for Selecting Test Harness Properties

Before selecting test harness properties, consider the following:

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

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

## Harness Name

Test harnesses must use valid MATLAB filenames.

## Save Test Harnesses Externally

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

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

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

See “Manage Test Harnesses” on page 2-32.

### Sources and Sinks

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

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

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

```
simulink/Sinks/Terminator
```

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

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

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

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

### Enable Initialize, Reset, and Terminate ports

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

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

## Add Separate Assessment Block

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

A Test Assessment block is a separate Test Sequence block configured with properties commonly used for verifying the component under test. For more information, see “Assessment Basics” on page 3-10 and “Activate verify Statements” on page 3-14.

## Open Harness After Creation

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

## Create without compiling the model

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.

## Create scalar inputs

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

## Post-create callback method

You can customize your test harness using a post-create callback. A post-create callback is a function that runs after the harness is created. For example, your callback can set up signal logging, add custom blocks, or change the harness simulation times. For more information, see “Customize Test Harnesses” on page 2-44.

### Rebuild harness on open

When you select this property, the test harness rebuilds every time you open it. For details on the rebuild process, see “Synchronize Changes Between Test Harness and Model” on page 2-57.

### Update Configuration Parameters and Model Workspace data on rebuild

When you select this property, configuration parameters and model workspace data update when you rebuild the harness. For details on the rebuild process, see “Synchronize Changes Between Test Harness and Model” on page 2-57.

### Post-rebuild callback method

You can customize your test harness using a post-rebuild callback. A post-rebuild callback is a function that runs after the harness is rebuilt. For example, your callback can set up signal logging, add custom blocks, or change the harness simulation times. For more information, see “Customize Test Harnesses” on page 2-44.

### Synchronization Mode

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

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

## Verification Modes

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

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

---

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

---

## See Also

Test Sequence | “Synchronize Changes Between Test Harness and Model” on page 2-57

## Test Harness Parameters and Signals

**In this section...**

“Test Harness Generation Without Compilation” on page 2-22

“Signal Conversion Subsystem” on page 2-22

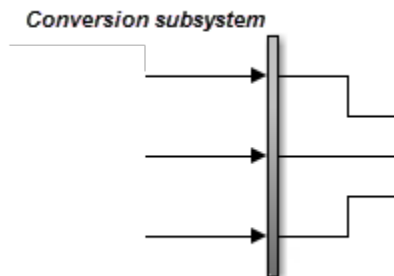
### Test Harness Generation Without Compilation

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 2-57.

### Signal Conversion Subsystem



A signal conversion subsystem

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

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

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

---

**Note** When you rebuild the test harness, the signal conversion subsystems are rebuilt. If you modify a conversion subsystem, disable automatic test harness rebuild to avoid losing your modifications when you open the test harness. See “Create Test Harnesses and Select Properties” on page 2-15.

---

## Refine, Test, and Debug a Subsystem

In this section...
“Model and Requirements” on page 2-24
“Create a Harness for the Controller” on page 2-26
“Inspect and Refine the Controller” on page 2-28
“Add Test Inputs and Test the Controller” on page 2-28
“Debug the Controller” on page 2-29

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'))
```

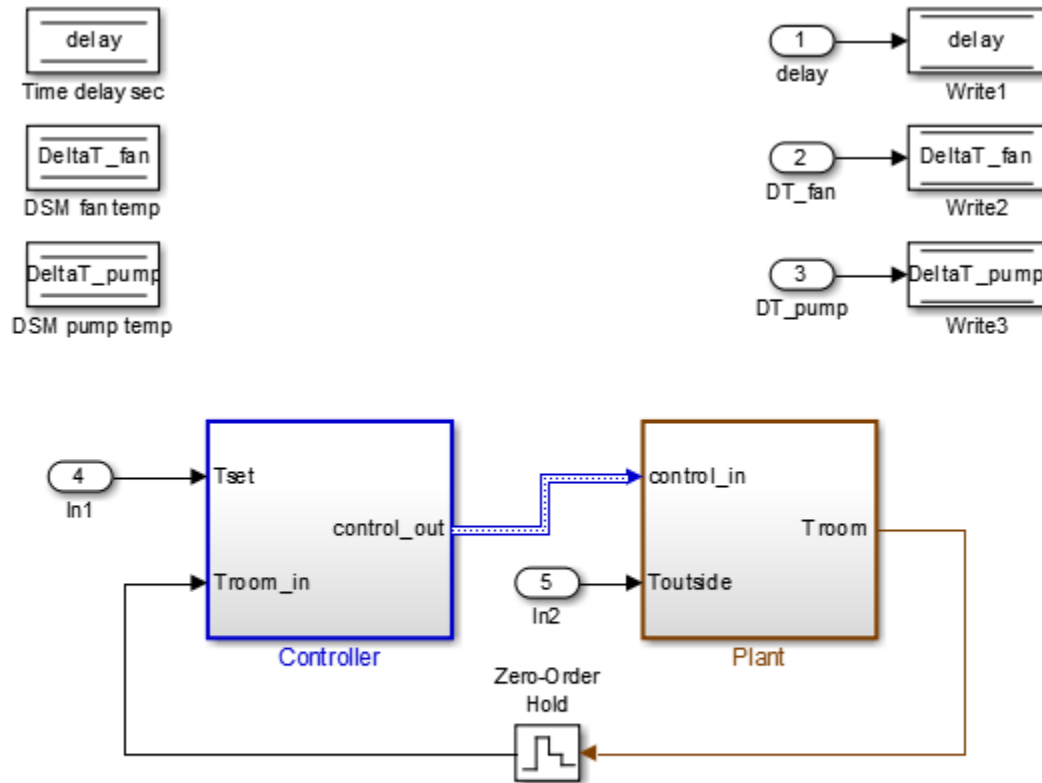
- 2 Copy this model file and supporting files to a writable location on the MATLAB path:

```
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.

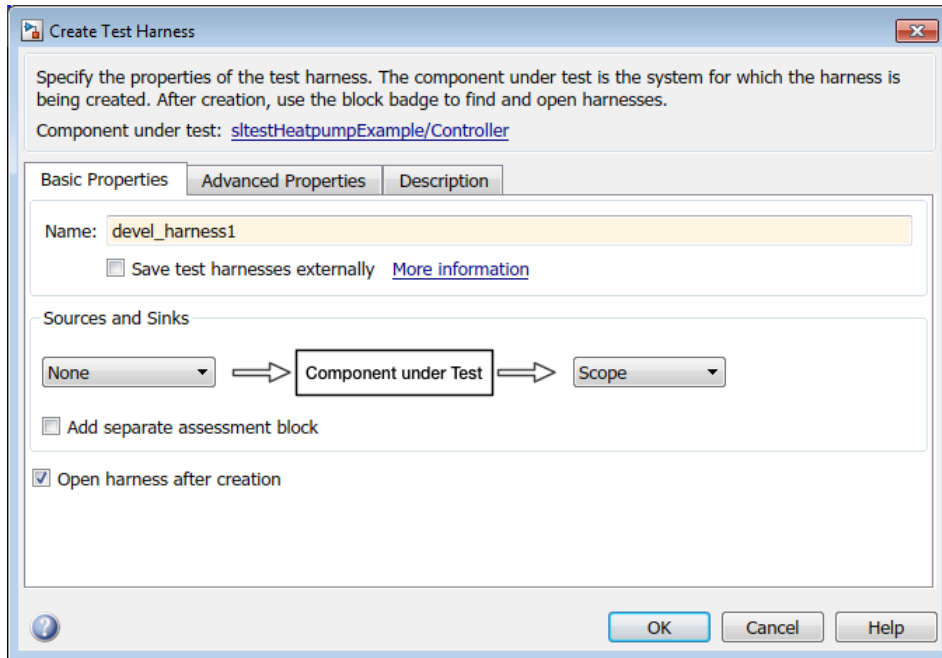
Temperature condition	System state	Fan command	Pump command	Pump direction
$ T_{room} - T_{set}  < \Delta T_{fan}$	idle	0	0	0
$\Delta T_{fan} \leq  T_{room} - T_{set}  < \Delta T_{pump}$	fan only	1	0	0
$ T_{room} - T_{set}  \geq \Delta T_{pump}$ and $T_{set} < T_{room}$	cooling	1	1	-1
$ T_{room} - T_{set}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

## Create a Harness for the Controller

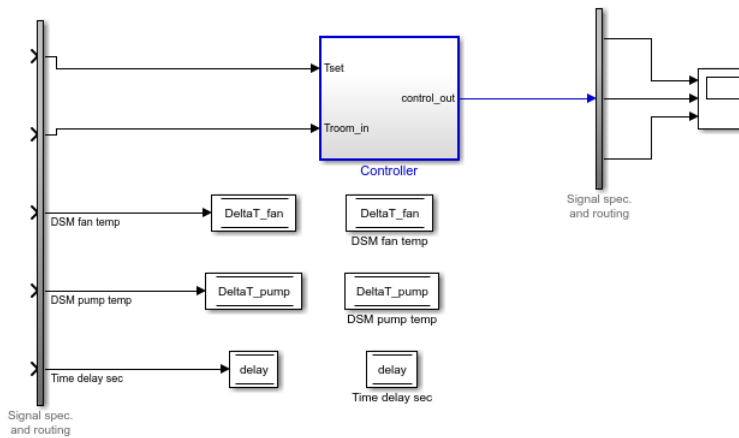
- 1 Right-click the Controller subsystem and select **Test Harness > Create for 'Controller'**.
- 2 Set the harness properties:

In the **Basic Properties** tab:

- **Name:** devel\_harness\_1
- Clear **Save test harness externally**
- **Sources and Sinks:** None and Scope
- Clear **Add separate assessment block**
- Select **Open harness after creation**

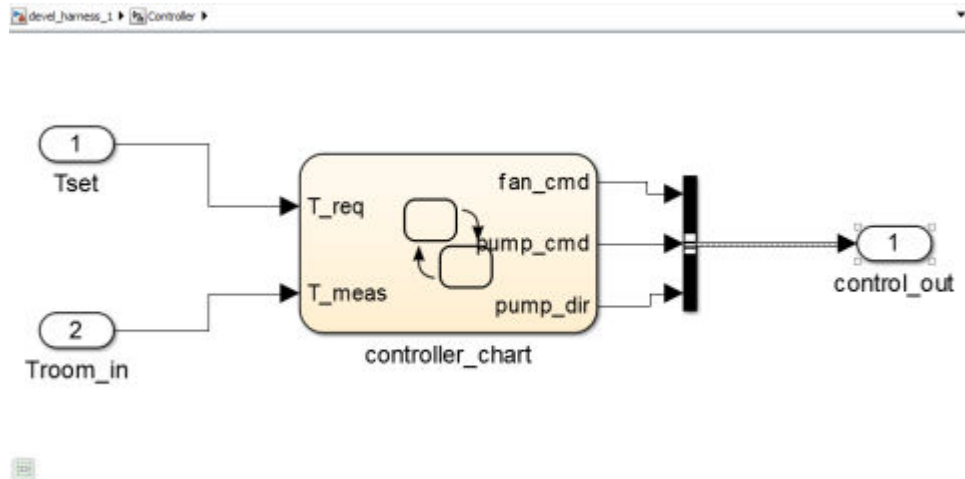


- 3 Click **OK** to create the test harness.



### Inspect and Refine the Controller

- 1 In the test harness, double-click Controller to open the subsystem.
- 2 Connect the chart to the Inport blocks.

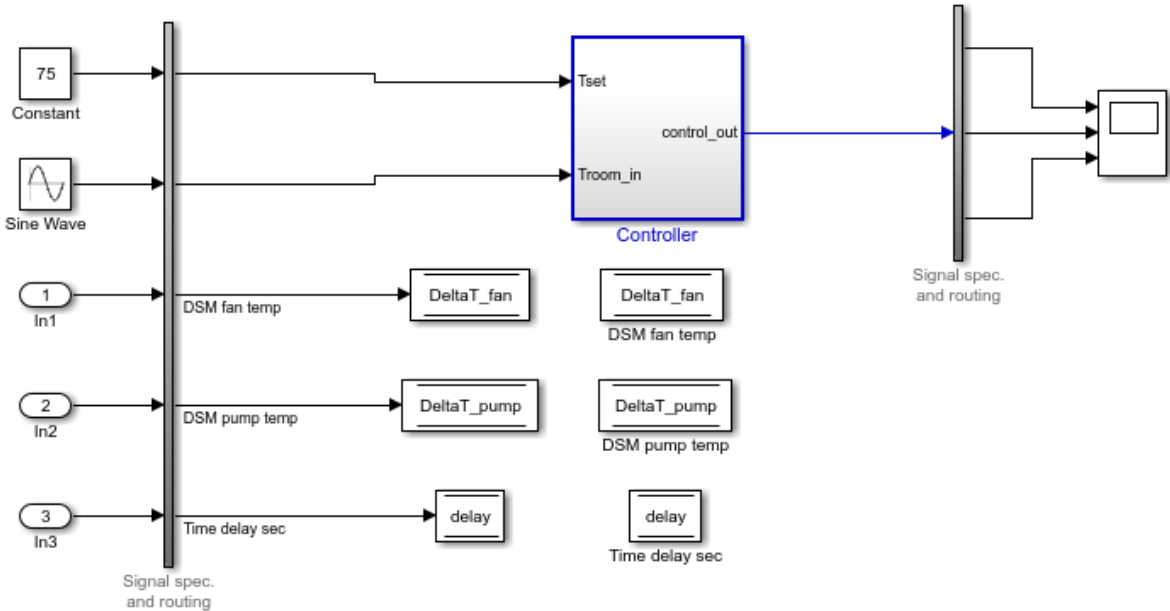


- 3 In the test harness, click the Save button to save the test harness and model.

### Add Test Inputs and Test the Controller

- 1 Navigate to the top level of `devel_harness_1`.
- 2 Create a test input for the harness with a constant `Tset` and a time-varying `Troom`. Connect a Constant block to the `Tset` input and set the value to 75.
- 3 Add a Sine Wave block to the harness model to simulate a temperature signal. Connect the Sine Wave block to the conversion subsystem input `Troom_in`.
- 4 Double-click the Sine Wave block and set the parameters:
  - **Amplitude:** 15
  - **Bias:** 75
  - **Frequency:**  $2 \cdot \pi / 3600$
  - **Phase (rad):** 0
  - **Sample time:** 1

- Select **Interpret vector parameters as 1-D**.
- 5 Connect Inport blocks to the Data Store Write inputs.

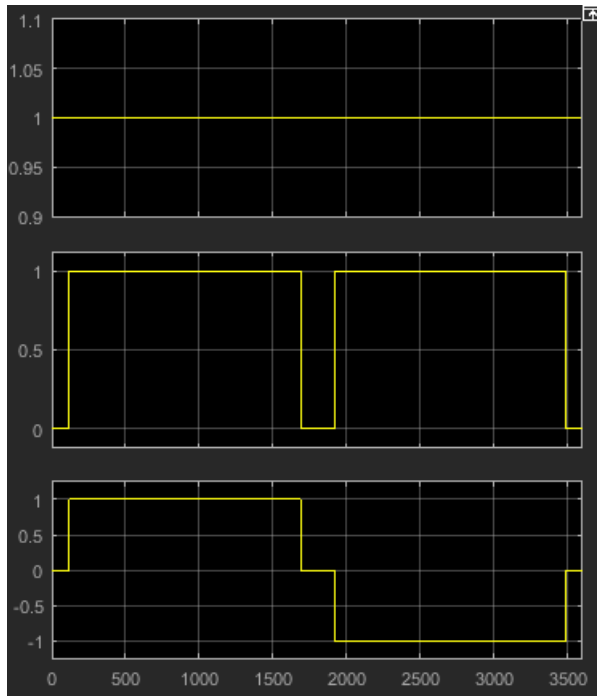


- 6 In the Configuration Parameters dialog box, in the **Data Import/Export** pane, select **Input** and enter `u`. `u` is an existing structure in the MATLAB base workspace.
- 7 In the **Solver** pane, set **Stop time** to 3600.
- 8 Open the scope in the test harness and change the layout to show three plots.
- 9 Click **Run** to simulate.

## Debug the Controller

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

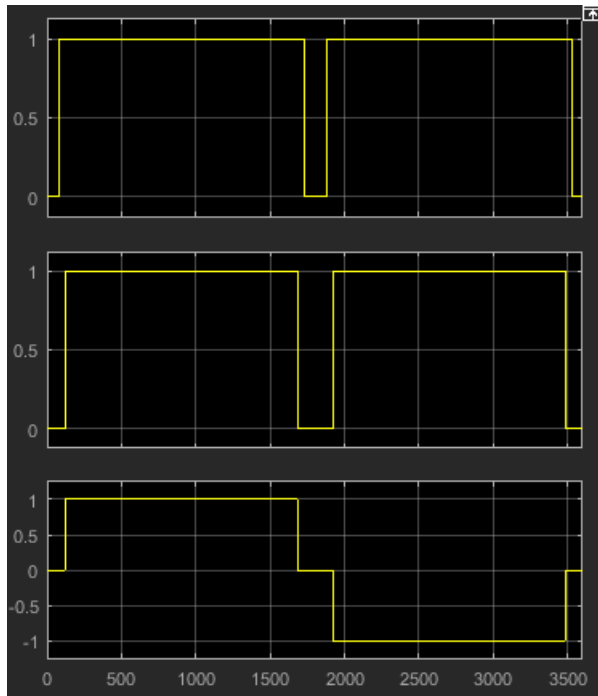
This is a bug. `fan_cmd` should equal 0 at IDLE. The `fan_cmd` control output must be changed for IDLE.



- 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.

## See Also

### Related Examples

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

## Manage Test Harnesses

In this section...
“Internal and External Test Harnesses” on page 2-32
“Manage External Test Harnesses” on page 2-32
“Convert Between Internal and External Test Harnesses” on page 2-34
“Preview and Open Test Harnesses” on page 2-35
“Find Test Cases Associated with a Test Harness” on page 2-36
“Export Test Harnesses to Separate Models” on page 2-36
“Clone and Export a Test Harness to a Separate Model” on page 2-37
“Delete Test Harnesses Programmatically” on page 2-39
“Move and Clone Test Harnesses” on page 2-41

### Internal and External Test Harnesses

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

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

See “Create Test Harnesses and Select Properties” on page 2-15.

### Manage External Test Harnesses

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



Follow these guidelines for external test harnesses:

---

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

---

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

Copying the model file on disk will not copy external harnesses associated with the model.

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

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

## Convert Between Internal and External Test Harnesses

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

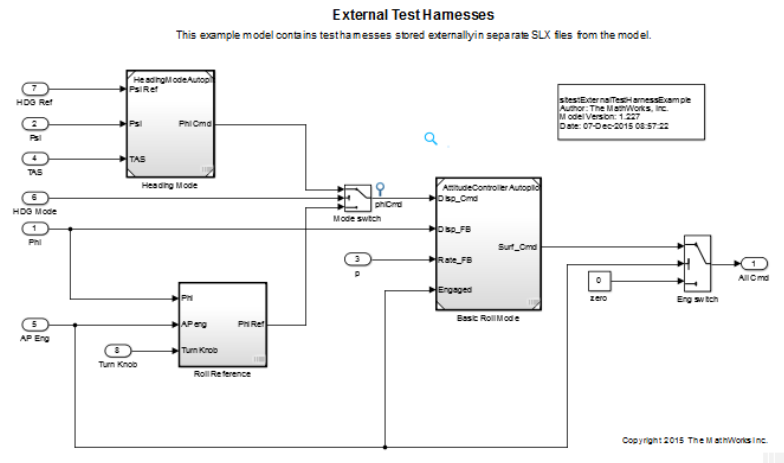
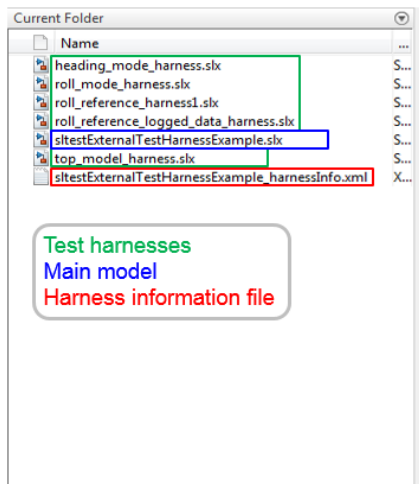
- Develop your model using internal test harnesses so that you can more easily share the model for review. When you complete your design and place the model under change control, convert to external harnesses.
- Use the change-controlled model as the starting point for a new design. Test the existing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

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

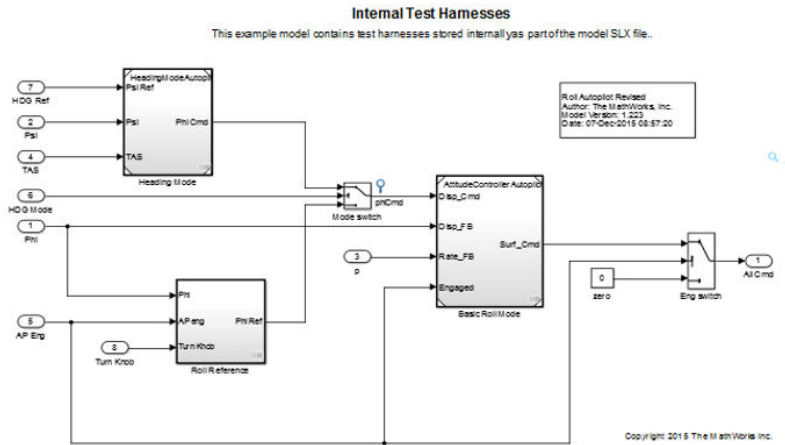
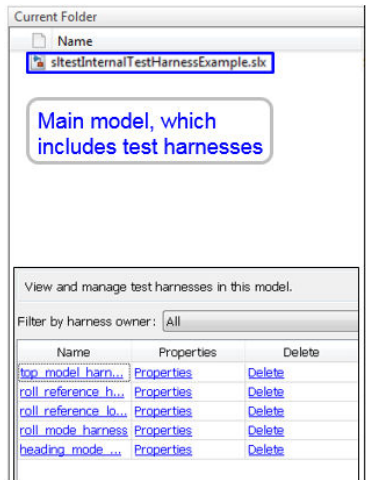
- 1 Navigate to the top of the main model.
- 2 Select **Analysis > Test Harness > Convert To External (Internal) Harnesses**.
- 3 A dialog box provides information on the conversion procedure and the affected test harnesses. Click **Yes** to continue.

The harnesses are converted.

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

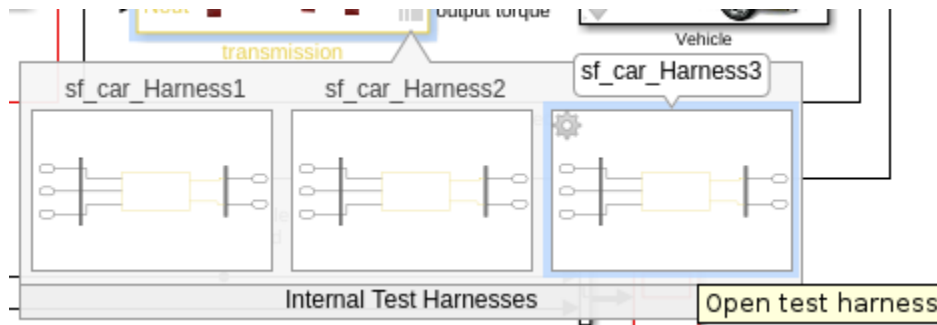


Inversely, conversion to internal test harnesses moves the test harness SLX files and the harnessInfo.xml file.

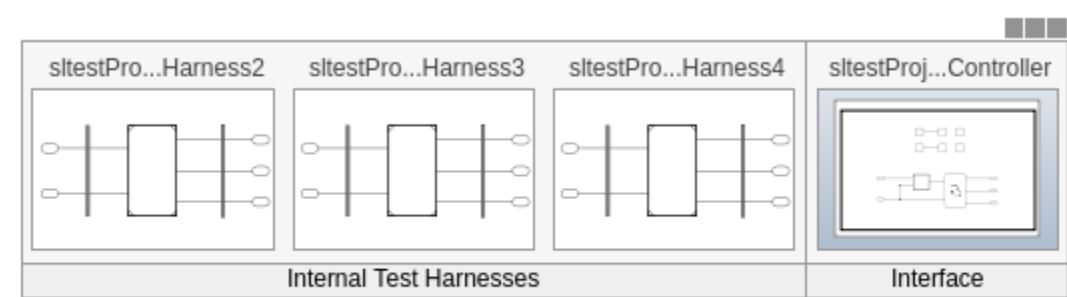


## Preview and Open Test Harnesses


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

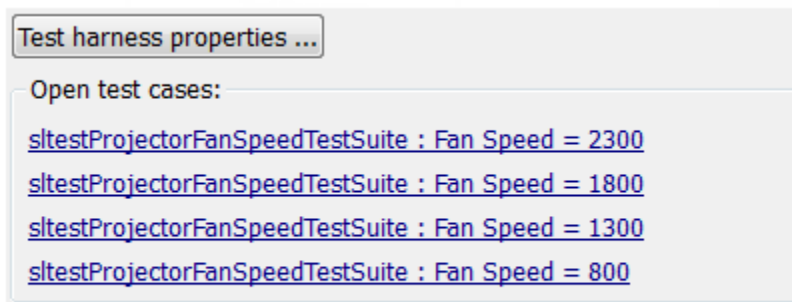


To view test harnesses for a model block diagram, click the pullout icon in the model canvas. To open a test harness, click a tile.



### Find Test Cases Associated with a Test Harness

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



### Export Test Harnesses to Separate Models

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

- To export an individual test harness:

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

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

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

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

See `sltest.harness.export`.

## Clone and Export a Test Harness to a Separate Model

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

### High-level Workflow

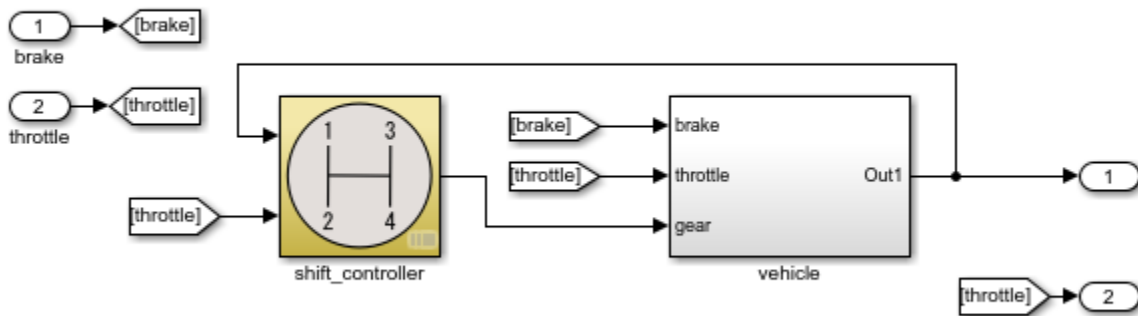
- 1 If you don't know the exact properties of the test harness you want to clone, get them using `sltest.harness.find`. You need the harness owner ID and the harness name.
- 2 Clone the test harness using `sltest.harness.clone`.
- 3 Export the test harness to a separate model using `sltest.harness.export`.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

### Open the Model and Save a Local Copy

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

### Testing Downshift Points of a Transmission Controller

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



Copyright 2016 The MathWorks, Inc.

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

### Get the Properties of the Source Test Harness

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

```
properties =
```

```
struct with fields:
```

```
    model: 'sltestTestSequenceExample'
    name: 'controller_harness'
  description: ''
    type: 'Testing'
  ownerHandle: 12.0018
  ownerFullPath: 'sltestTestSequenceExample/shift_controller'
    ownerType: 'Simulink.SubSystem'
    isOpen: 0
  canBeOpened: 1
```

```

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

```

### Clone the Test Harness

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

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

### Save the Model

Before exporting the harness, save changes to the model.

```
save_system(model)
```

### Export the Test Harness to a Separate Model

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

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

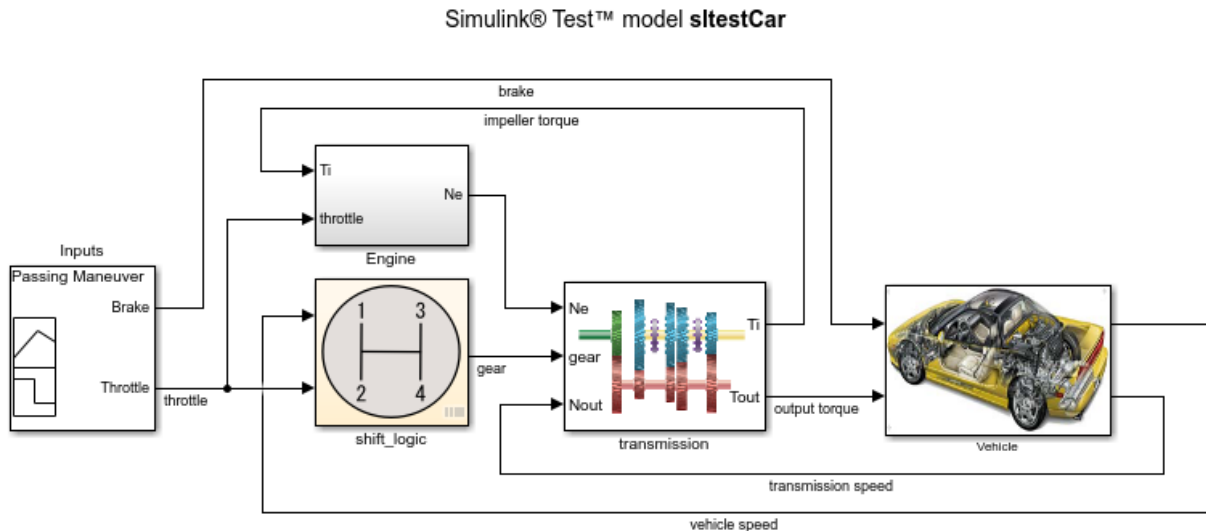
clear('model')
clear('properties')
close_system('sltestTestSequenceExample',0)
```

### Delete Test Harnesses Programmatically

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

1. Open the model

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



Copyright 1997-2017 The MathWorks, Inc.

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

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

3. Find the harnesses in the model.

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

```
test_harness_list =
```

```
1x5 struct array with fields:
```

```
model
name
```



```

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

```

4. Delete the harnesses.

```

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

close_system('sltestCar',0);

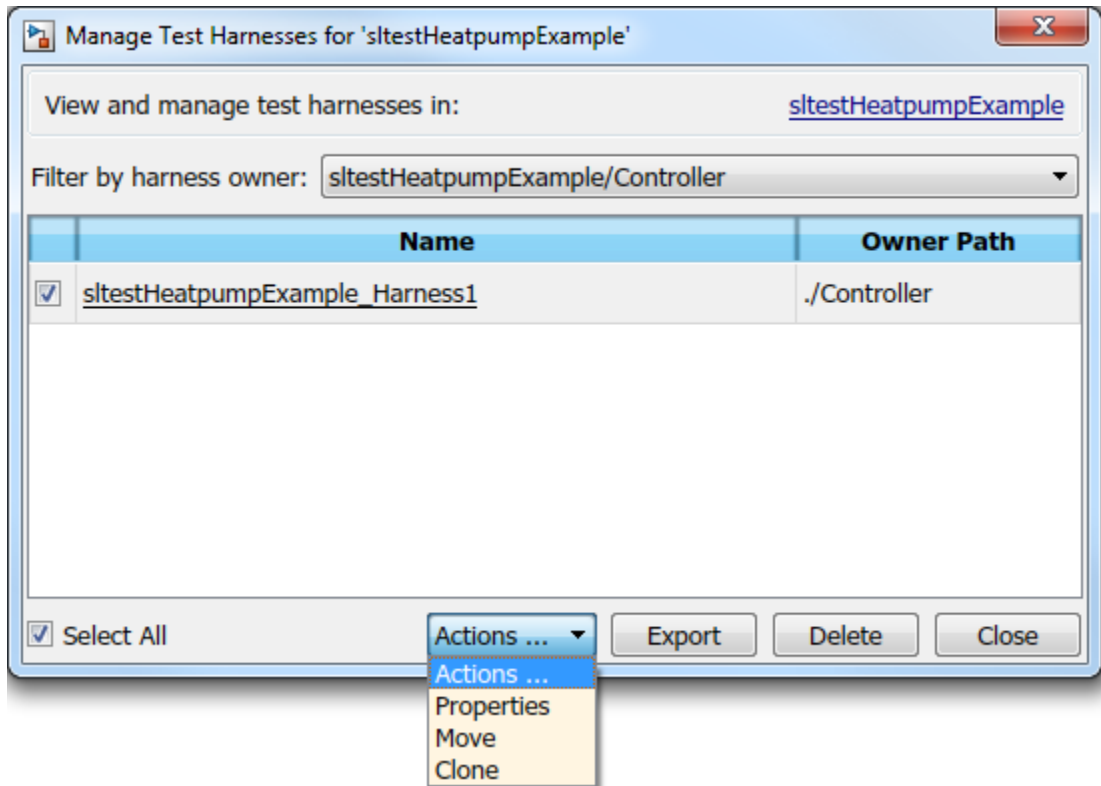
```

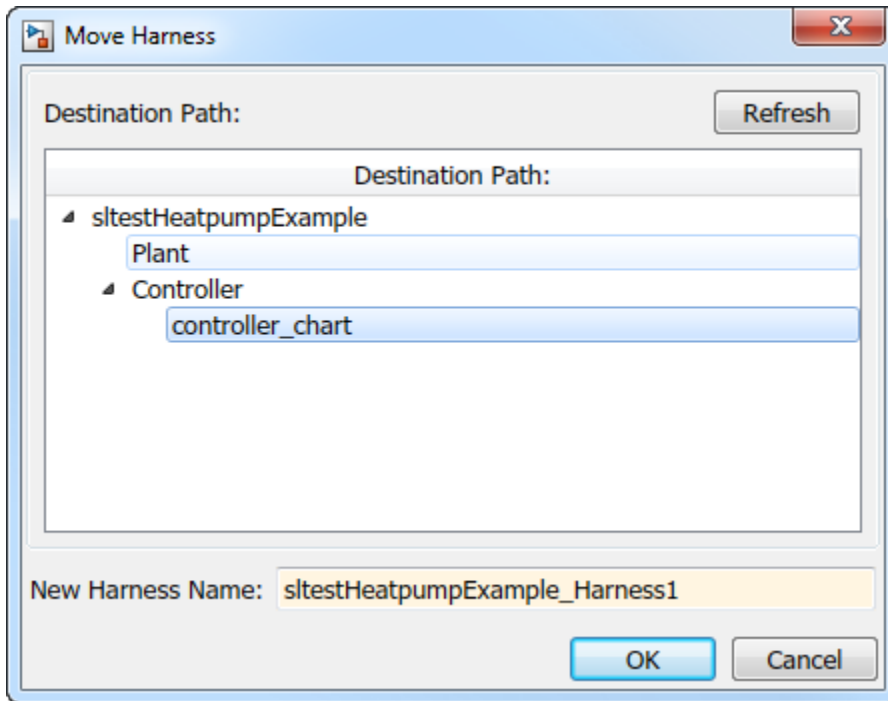
## Move and Clone Test Harnesses

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

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

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





Select the destination path and name your test harness.

## See Also

### Functions

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

## Customize Test Harnesses

In this section...
“Callback Function Definition and Harness Information” on page 2-45
“How to Display Harness Information struct Contents” on page 2-47
“Customize a Test Harness to Create Mixed Source Types” on page 2-47
“Test Harness Callback Example” on page 2-49

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

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

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

- 1 Create the callback function.
- 2 In the function, use the Simulink programmatic interface to script the commands to customize the test harness. For more information, see the functions listed in “Model Editing Fundamentals” (Simulink).
- 3 Specify the function as the post-create or post-rebuild callback:
  - For a new test harness,
    - If you are using the UI, enter the function name in the **Post-create callback method** or **Post-rebuild callback method** in the **Advanced Properties** of the harness creation dialog box.
    - If you are using `sltest.harness.create`, specify the function as the `PostCreateCallback` or `PostRebuildCallback` value.
  - For an existing test harness,
    - If you are using the UI, enter the function name in **Post-rebuild callback method** in the harness properties dialog box.

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

For more information on test harness properties, see “Create Test Harnesses and Select Properties” on page 2-15.

## Callback Function Definition and Harness Information

The callback function declaration is

```
function myfun(x)
```

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

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

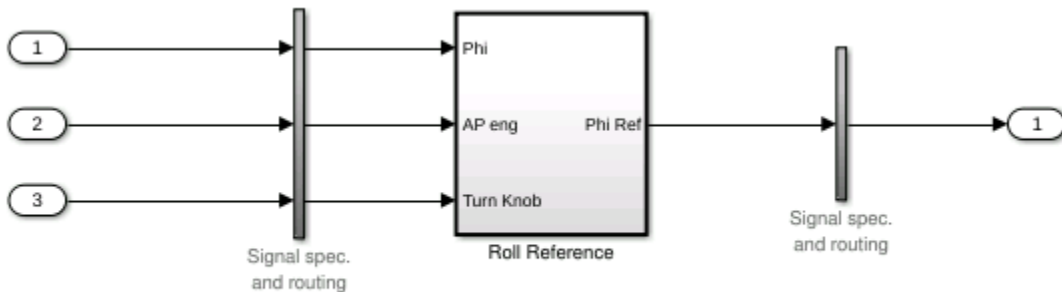
```
function harnessCustomization(harnessInfo)
```

```
% Script commands here to customize your test harness.
```

```
end
```

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

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



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

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

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

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

```
harnessName = get_param(harnessInfo.HarnessModel, 'Name');  
block = add_block('simulink/Sources/Constant', [harnessName '/ConstInput']);
```

- To get the port handles for the component under test, get the `'PortHandles'` parameter for `harnessInfo.HarnessCUT`.

```
CUTPorts = get_param(harnessInfo.HarnessCUT, 'PortHandles');
```

- To get the simulation stop time for the test harness, get the `'StopTime'` parameter for `harnessInfo.HarnessModel`.

```
st = get_param(harnessInfo.HarnessModel, 'StopTime');
```

- To set a 16 second simulation stop time for the test harness, set the `'StopTime'` parameter for `harnessInfo.HarnessModel`.

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

## How to Display Harness Information struct Contents

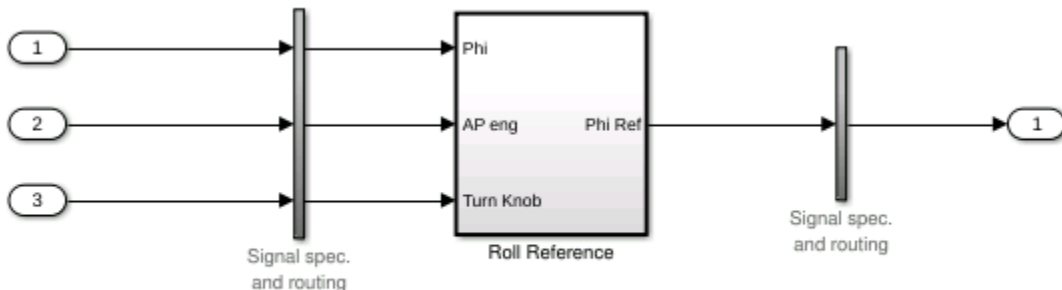
To list the harness information for your test harness:

- 1 In the callback function, add the line
 

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

## Customize a Test Harness to Create Mixed Source Types

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



The function follows the procedure:

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

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

```
function harnessCustomization(harnessInfo)

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

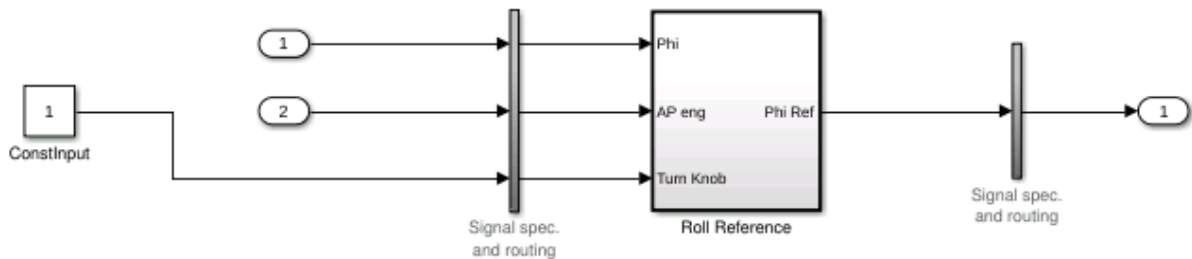
% Add Constant block:
constBlock = add_block('simulink/Sources/Constant',[harnessName '/ConstInput']);

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

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

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

end
```





## Test Harness Callback Example

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

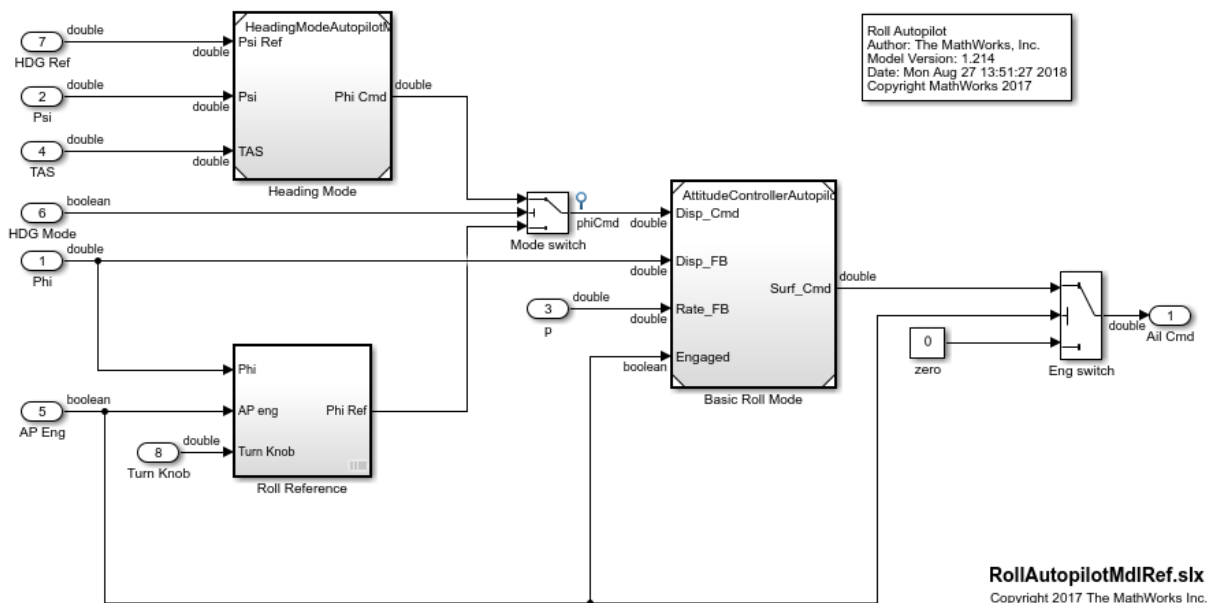
### The Model

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

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

#### Requirements-based Testing for Controller Development

This model is used to show how to perform requirements-based testing using test harnesses, Test Sequence blocks, and the test manager.  
To view the demo, enter `sltestRequirementsTestingAutopilotDemo` in MATLAB(R).



### Get Path to the Harness Customization Function

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

### The Customization Function and Test Harness Information

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

```
type(cbFile)
```

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

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

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

### Create the Customized Test Harness

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

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

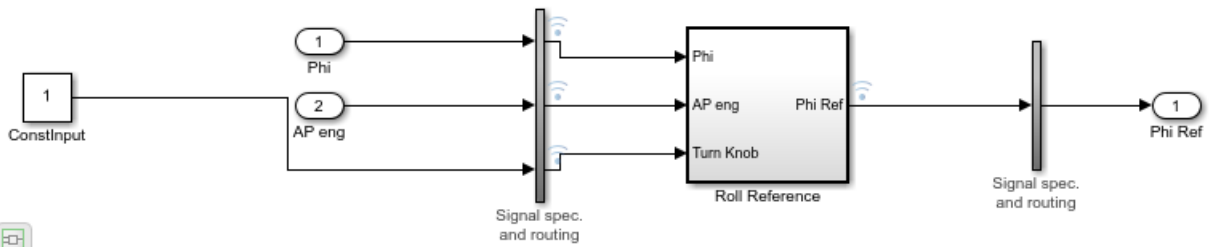
2. In the `RollAutopilotMdlRef` model, right-click the `Roll Reference` subsystem and select **Test Harness > Create for Roll Reference**.

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

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

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

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



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

## See Also

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

## Related Examples

- “Create Test Harnesses and Select Properties” on page 2-15

## Create Test Harnesses from Standalone Models

<b>In this section...</b>
“Test Harness Import Workflow” on page 2-52
“Component Compatibility for Test Harness Import” on page 2-53
“” on page 2-54

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

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

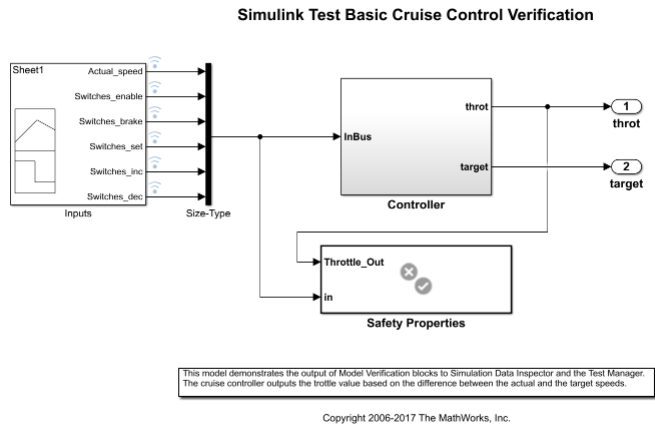
A common test model passes input signals to a copy of a subsystem or a Model block referencing your main model. Test models include models created by Simulink Coverage™ and Simulink Design Verifier™.

### Test Harness Import Workflow

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

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

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



## Component Compatibility for Test Harness Import

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

### Component Compatibility for Test Harness Import

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

You cannot create a test harness by importing:

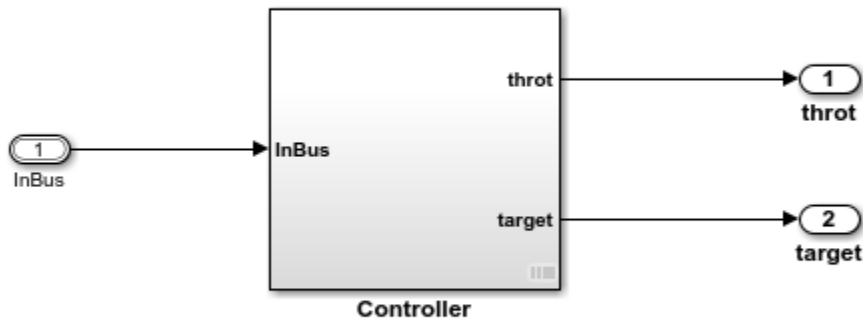
- Libraries
- Models that have existing test harnesses
- Models with unsaved changes. Save open models before importing

### Import a Standalone Model as a Test Harness

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

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

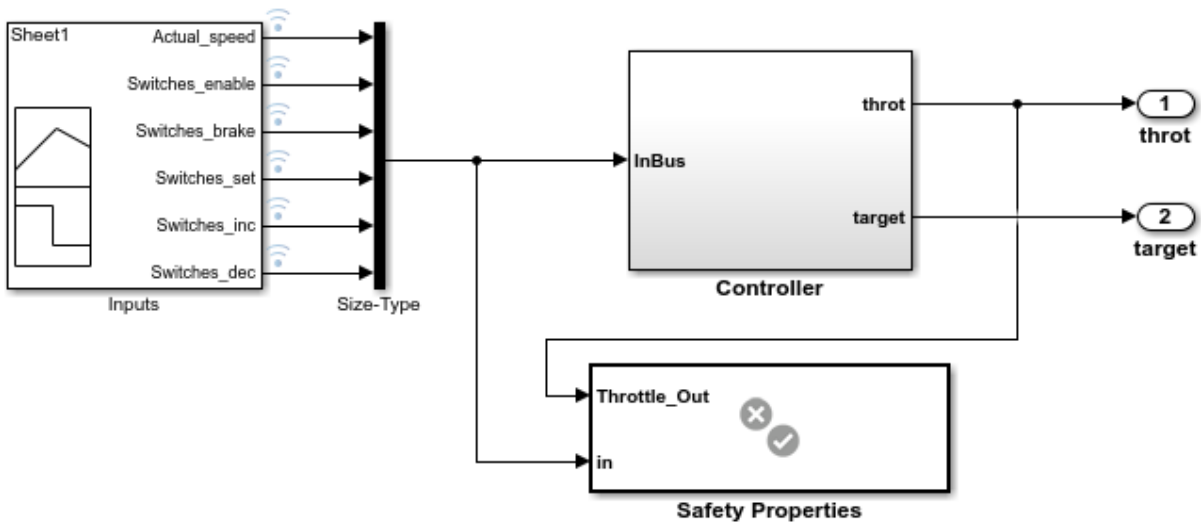
Simulink Test: Basic Cruise Control



Copyright 2006-2018 The MathWorks, Inc.

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

## Simulink Test Basic Cruise Control Verification

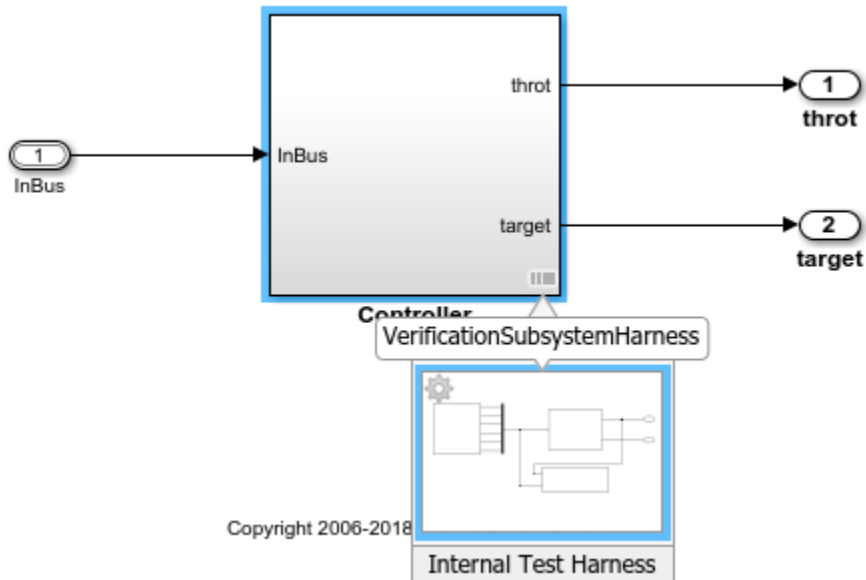


Copyright 2006-2018 The MathWorks, Inc.

**Create a Test Harness from the Standalone Model**

1. In the main model, right-click the Controller subsystem and select **Test Harness > Import for 'Controller'**.
2. Set the following harness properties:
  - **Name:** VerificationSubsystemHarness
  - **Simulink model to import:** Click **Browse** and select sltestBasicCruiseControlHarnessModel in the MATLAB® examples/simulinktest directory.
  - **Component under Test in imported model:** Controller
3. Click **OK**.

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



## See Also

`sltest.harness.import`

## Related Examples

- “Test Harness and Model Relationship” on page 2-2
- “Synchronize Changes Between Test Harness and Model” on page 2-57



## Synchronize Changes Between Test Harness and Model

### In this section...

“Set Synchronization for a New Test Harness” on page 2-57

“Change Synchronization of an Existing Test Harness” on page 2-58

“Synchronize Configuration Set and Model Workspace Data” on page 2-58

“Check for Unsynchronized Component Differences” on page 2-59

“Rebuild a Test Harness” on page 2-59

“Push Changes from Test Harness to Model” on page 2-60

“Check Component and Push Parameter to Main Model” on page 2-60

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

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

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

### Set Synchronization for a New Test Harness

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

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

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

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

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

---

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

---

---

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

---

## Change Synchronization of an Existing Test Harness

To change a test harness synchronization mode:

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

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

## Synchronize Configuration Set and Model Workspace Data

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

## Check for Unsynchronized Component Differences

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

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

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

Consider these conditions when checking for unsynchronized differences:

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

## Rebuild a Test Harness

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

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

For more information, see “Create Test Harnesses and Select Properties” on page 2-15 and `sltest.harness.rebuild`.

## Push Changes from Test Harness to Model

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

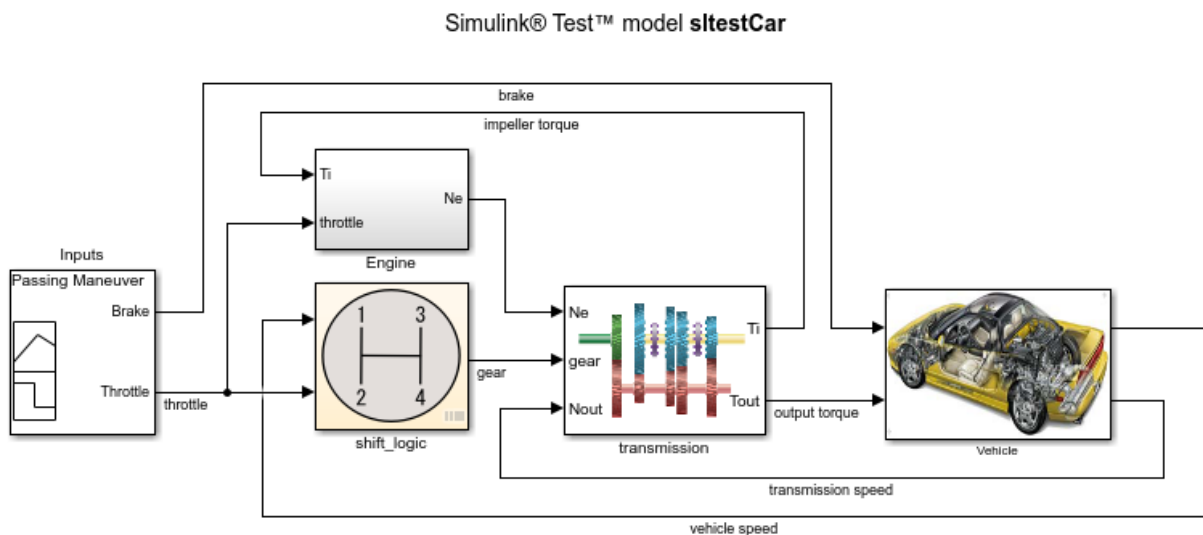
## Check Component and Push Parameter to Main Model

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

This example also includes programmatic steps.

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

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

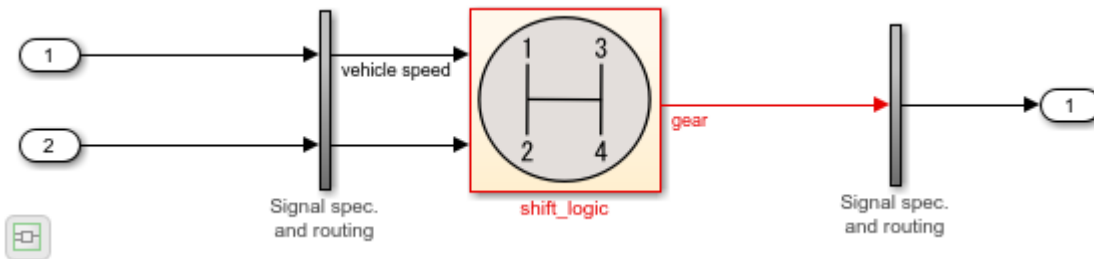


Copyright 1997-2017 The MathWorks, Inc.

## Update the Mask Parameter in the Test Harness

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

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



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

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

## Check Synchronization between Test Harness and Main Model

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

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

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

```
comparison
```

```
comparison =
```

```
logical
```

```
0
```

```
details
```

```
details =
```

```
  struct with fields:
```

```
    overall: 0
```

```
    contents: 1
```

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

### Update the Parameter to the Main Model

1. In the test harness, select **Analysis > Test Harness > Push Component and Parameters to Main Model**.
2. In the main model, double-click the shift\_logic subsystem. The parameter value is updated.

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

### Re-check Synchronization between Test Harness and Main Model

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

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

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

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

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

```
comparison
```

```
comparison =
```

```
  logical
```

```
  1
```

details

```
details =
```

```
  struct with fields:
```

```
    overall: 1
```

```
    contents: 1
```

```
    reason: 'The checksum of the harnessed component and the component in the main m
```

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

## See Also

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

## Related Examples

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

## Test Library Blocks

<b>In this section...</b>
---------------------------

“Library Testing Workflow” on page 2-64
---

“Library and Linked Subsystem Test Harnesses” on page 2-64
--

“Edit Library Block from a Test Harness” on page 2-66
---

“Testing a Library and a Linked Block” on page 2-66
---

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

### Library Testing Workflow

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

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

### Library and Linked Subsystem Test Harnesses

A test harness for a library subsystem has specific properties:


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

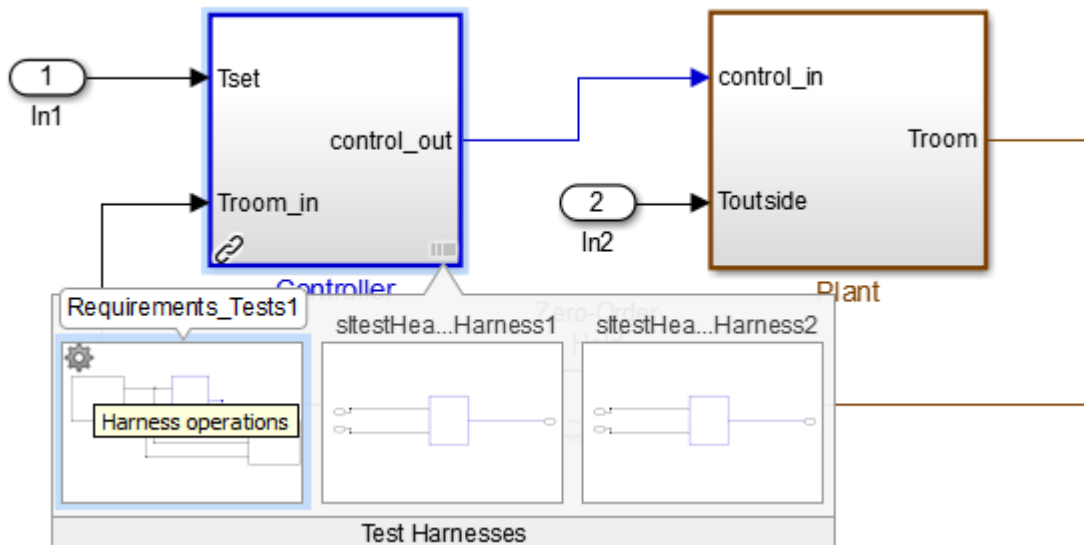


- A test harness for a library subsystem does not generate conversion subsystems for the block inputs and outputs.
- A library subsystem test harness does not use push or rebuild operations, because libraries do not use configuration parameters.

When you create a linked subsystem from a library subsystem, test harnesses copy to the linked instance. If you do not need the test harnesses, you can delete them. For instructions on deleting all test harnesses from a model, see “Manage Test Harnesses” on page 2-32.

When you create a test harness for a linked subsystem, the harness associates with the linked subsystem, not the library subsystem. You can move a test harness from a linked subsystem to the library subsystem. For example, this linked subsystem Controller has three test harnesses. To move the Requirements\_Tests1 test harness to the library:

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



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

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

### **Edit Library Block from a Test Harness**

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

If you have a library block whose design is complete, set your test harnesses to prevent changes to the component under test. You can set this property when you create the test harness or after harness creation. See “Create Test Harnesses and Select Properties” on page 2-15.

### **Testing a Library and a Linked Block**

Verify a reusable subsystem in a library and in a larger system. Copyright 2016 The MathWorks, Inc.

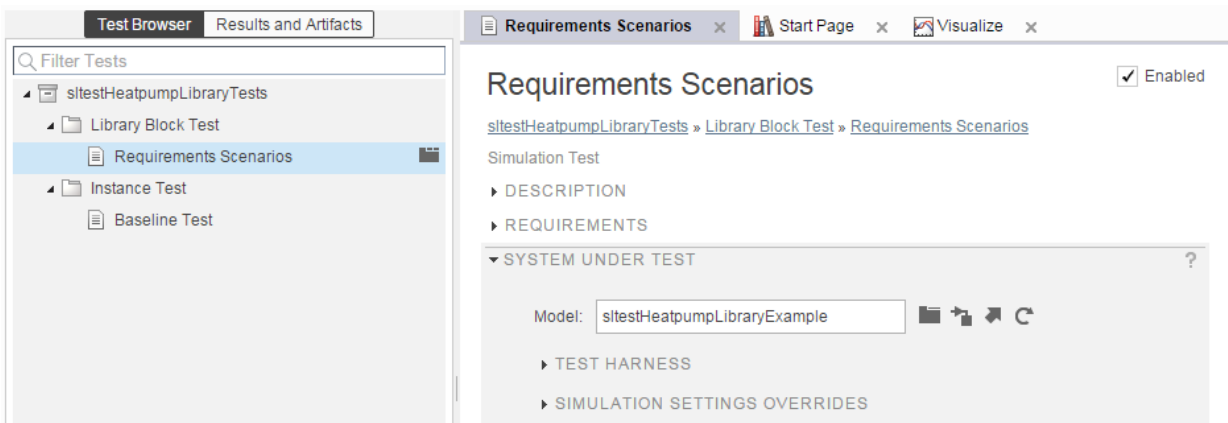
This example demonstrates a test case that confirms a library block meets a short set of requirements. After testing the library block, you execute a baseline test of a linked block and capture the baseline results. You then promote the baseline test harness to the library.

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

#### **Open the Test File**

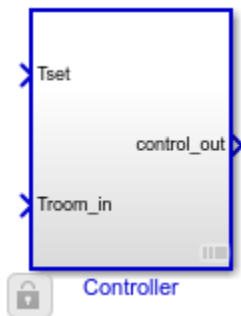
Enter the following to store paths and filenames for the example, and to open the test file. The test file contains a test case for the library block and for the block instance in a closed-loop model.

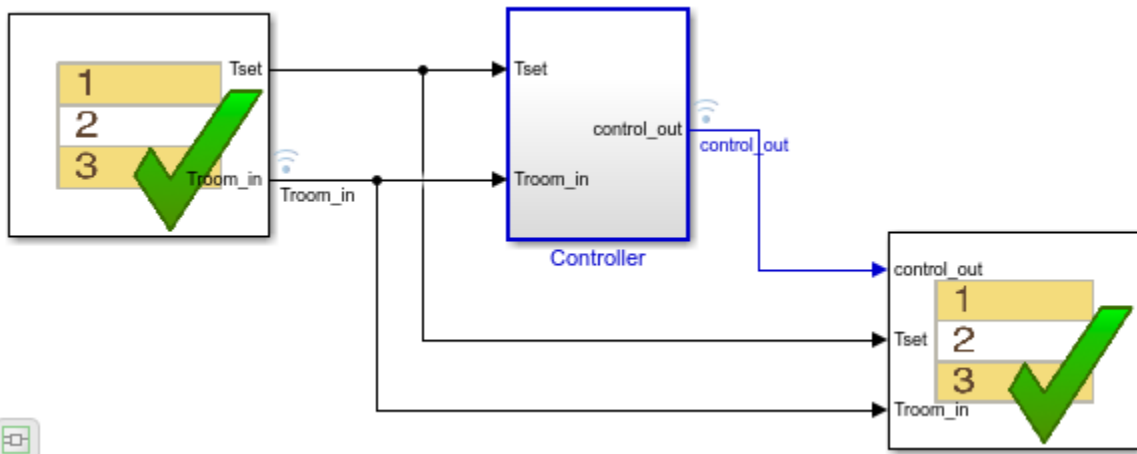
```
filePath = fullfile(matlabroot, 'toolbox', 'simulinktest', 'simulinktestdemos');  
testFile = 'sltestHeatpumpLibraryTests.mldatx';  
library = 'sltestHeatpumpLibraryExample';  
system = 'sltestHeatpumpLibraryLinkExample';  
open(fullfile(filePath, testFile));
```



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

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





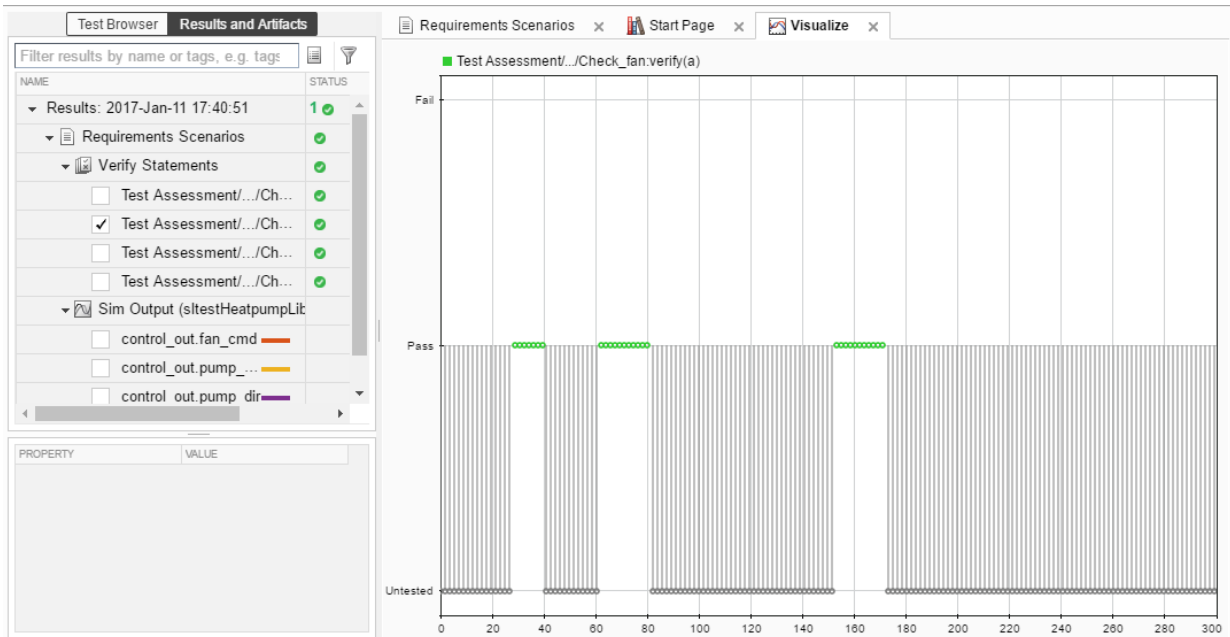
The Test Sequence block sets three scenarios for the controller:

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

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

### Run the Requirements-Based Test

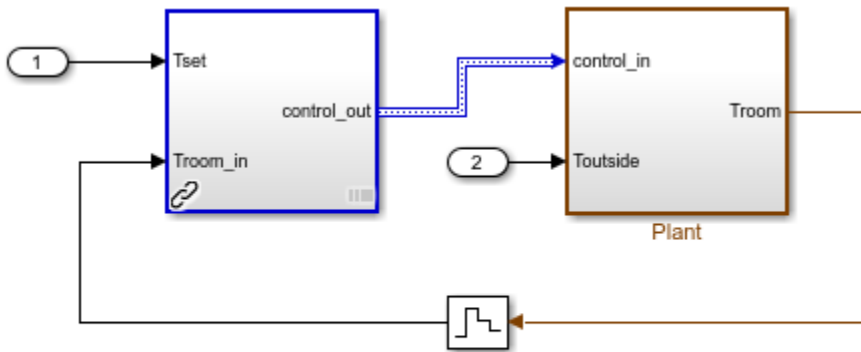
In the test manager, run the Requirements Scenarios test case. The `verify` statement results show that the `control_out` signals pass.



### Open the Linked Block Model

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

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

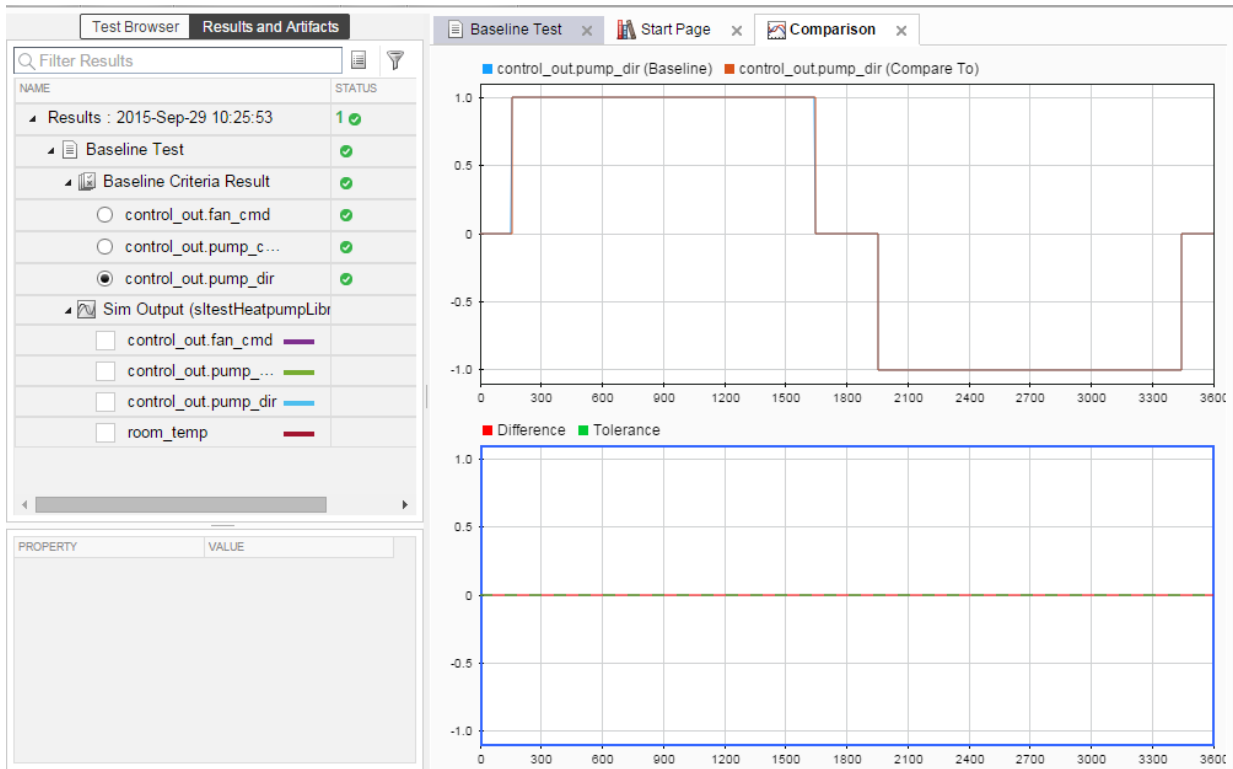


Copyright 1990-2014 The MathWorks, Inc.

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

### Run the Baseline Test and Observe Results

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



## Move the Test Harness to the Library

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

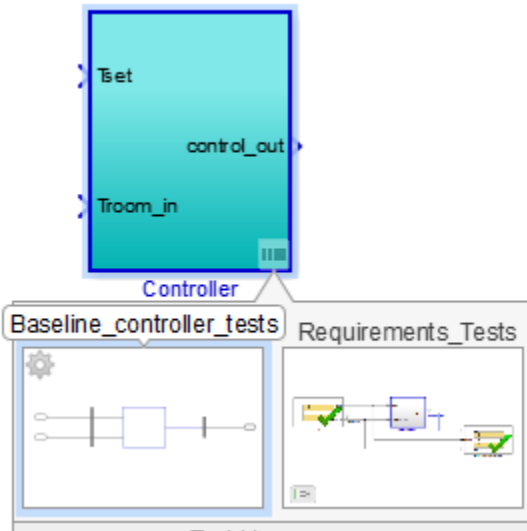
Move the **Baseline\_controller\_tests** test harness to the library block:

1. In the sltestHeatpumpLibraryLinkExample model, click the harness badge and hover over the **Baseline\_controller\_tests** test harness.
2. Click the harness operations icon



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

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



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

## See Also

### Related Examples

- “Testing a Library and a Linked Block”



# Test Sequences and Assessments

---

- “Test Sequence Basics” on page 3-2
- “Assessment Basics” on page 3-10
- “Activate verify Statements” on page 3-14
- “Test Sequence Editor” on page 3-20
- “Test Step Actions and Transitions” on page 3-32
- “Signal Generation Functions” on page 3-41
- “Run-Time Assessments” on page 3-49
- “Programmatically Create a Test Sequence” on page 3-56
- “Syntax for Test Sequences and Assessments” on page 3-60
- “Debug a Test Sequence” on page 3-69
- “Test Downshift Points of a Transmission Controller” on page 3-72
- “Test Assessment Reuse” on page 3-79
- “View Graphical Results From Model Verification Library” on page 3-84

## Test Sequence Basics

### In this section...

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

“Test Sequence Hierarchy” on page 3-2

“Transition Types” on page 3-2

“Create a Basic Test Sequence” on page 3-4

“Create Basic Test Assessments” on page 3-5

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. A test step contains actions that execute at the beginning of the step. A test step can also contain transitions that define when the step stops executing, and which test step executes next. Action and transition syntax uses MATLAB as the action language.

### Test Sequence Hierarchy

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

### Transition Types

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

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

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

Step	Transition	Next Step
<b>PressNeitherButton</b> RedButtonIN = false; GreenButtonIN = false;	1. after(1,sec)	PressBothButtons ▼
<b>PressBothButtons</b> RedButtonIN = true; GreenButtonIN = true;	1. after(1,sec)	PressRedButton ▼
<b>PressRedButton</b> RedButtonIN = true; GreenButtonIN = false;	1. after(1,sec)	PressGreenButton ▼
<b>PressGreenButton</b> RedButtonIN = false; GreenButtonIN = true;	1. after(1,sec)	EndTest ▼
<b>EndTest</b>		

- When decomposition:** When decomposition sequences are analogous to switch statements in programming. Your sequence can act based on specific conditions occurring in your model. In a When decomposition sequence, steps activate based on a condition that you define after the step name. Transitions are not used between steps. This sequence is a series of assessments. Each assessment is active when the signal gear is equal to a different value. For more information, see “Test Sequence Editor” on page 3-20.

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

## Create a Basic Test Sequence

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

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

```
openExample('simulinktest/TransmissionDownshiftTestSequenceExample')
```
- 2 Right-click the `shift_controller` subsystem and select **Test Harness > Create for 'shift\_controller'**.
- 3 In the Create Test Harness dialog box, under **Sources and Sinks**:
  - Under **Sources and Sinks**, select Test Sequence from the source drop-down menu.
  - Under **Sources and Sinks**, select **Add separate assessment block**.
  - Select **Open harness after creation**.
- 4 Click **OK**. The test harness for the `shift_controller` subsystem opens.
- 5 Double-click the Test Sequence block. The Test Sequence Editor opens.

Step	Transition	Next Step	Description
Run %% Initialize data outputs. speed = 0; throttle = 0;			

- 6 Create the test sequence.
  - a Rename the first step Accelerate and add the step actions:
 

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

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

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

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

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

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

- d** Add a constant to define Limit. In the **Symbols** pane, hover over **Constant** and click the add data button. Enter Limit for the constant name.
- e** Hover over Limit and click the edit button. In the **Constant value** field, enter 2. Click **OK**.

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	<b>Stop</b> throttle = 0; speed = 0;		
<b>Local</b>			
<b>Constant</b> Limit			
<b>Parameter</b>			
<b>Data Store Memory</b>			

## Create Basic Test Assessments

- Continuing the example, in the test harness, double-click the Test Assessment block to open the editor. The editor displays a When decomposition sequence.
- Rename the first step Assessments.
- Add two steps to Assessments. Right-click the Assessments step and select **Add sub-step**. Do this a second time. There should be four steps under Assessments.

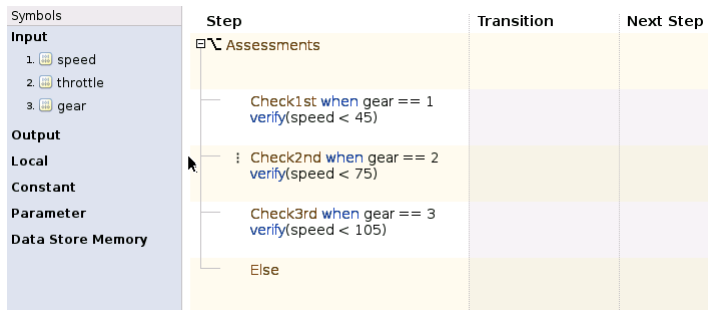
- 4 Enter the names and actions for the four substeps.

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

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

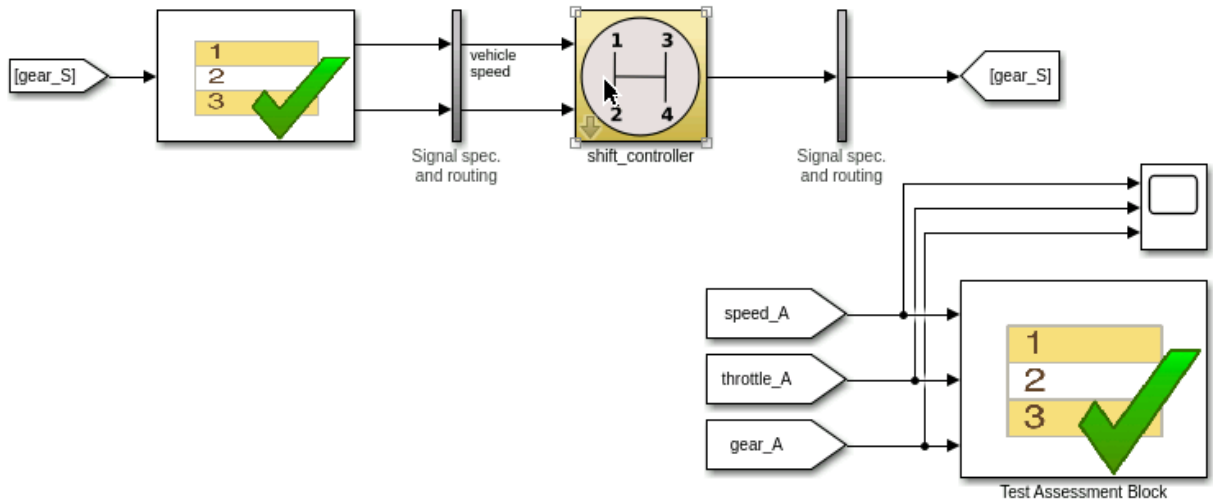
Check3rd when gear == 3  
verify(speed < 105)

Else

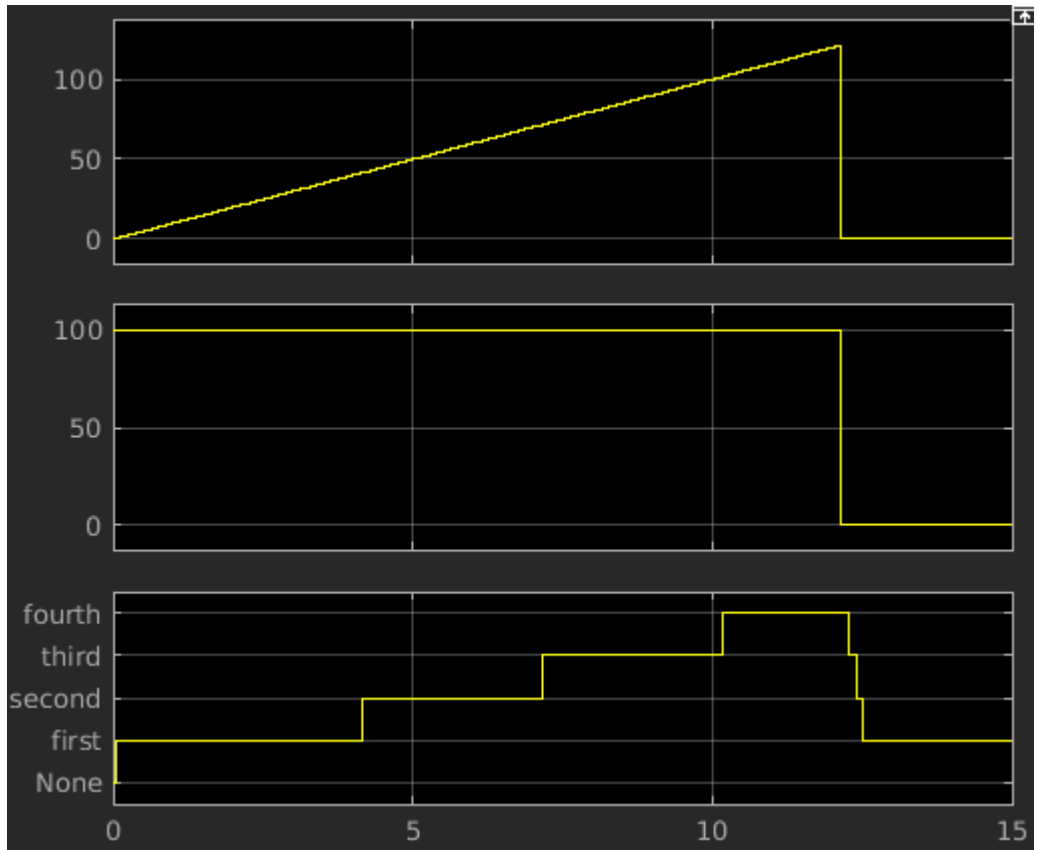


The fourth step Else has no actions. Else handles simulation conditions outside of the preceding when conditions.

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

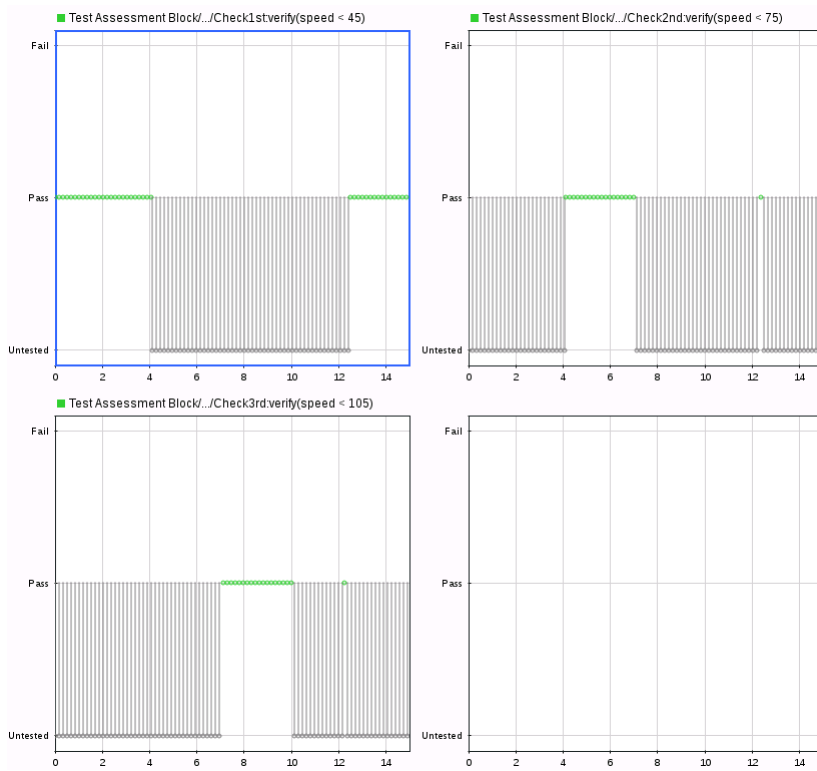


- 6 Set the model simulation time to 15 seconds and simulate the test harness. View the signal data by opening the scope.



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





## See Also

“Test Sequence Editor” on page 3-20 | “Syntax for Test Sequences and Assessments” on page 3-60 | Test Sequence

# Assessment Basics

In this section...
“Overview” on page 3-10
“Comparing Simulation to Baseline Data or Another Simulation” on page 3-11
“Post-Processing Simulation Output” on page 3-11
“Run-Time Assessments” on page 3-12

## Overview

Functional testing requires assessing simulation behavior and comparing simulation output to expected output. Assessments are used, for example, to:

- Compare a final signal value to a constant.
- Compare timeseries data to a baseline.
- Find peaks in timeseries data, and compare the peaks to a pattern.
- Analyze signal behavior in a time interval after an event.

This topic provides an overview of assessments, so that you can set up tests for your particular application. In the topic, you can find links to more detailed examples of each assessment.

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

- In the test case, you can:
  - Compare simulation output to baseline data.
  - Compare the output of two simulations.
  - Post-process simulation output using a custom script.
- In a test harness or model, you can:
  - Perform run-time assessments, which return a `pass`, `fail`, or `untested` result for each time step.
  - Use run-time assertions to stop simulation on a failure.

## Comparing Simulation to Baseline Data or Another Simulation

Baseline criteria compares simulation data to baseline data, according to tolerances. Equivalence criteria compares data from two simulations, according to tolerances. You can set baselines for numeric, enumerated, or logical data.

Set a numeric range for the comparison using absolute or relative tolerances. Set a time range for the comparison using leading and lagging tolerances. For numeric data, you can specify a comparison range using absolute tolerance, relative tolerance, leading tolerance, or lagging tolerance. For enumerated or logical data, you can specify a comparison range using leading or lagging tolerance. Results outside the tolerance range fail. For more information, see “Apply Tolerances to Test Criteria” on page 6-88.

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

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

SIGNAL NAME	ABS TOL	REL TOL	LD TOL	LG TOL
▼ <input type="checkbox"/> BrakeThrottleBaseline3.mat	0	0.10%	0	0
<input checked="" type="checkbox"/> output torque	0	0.10%	0	0
<input checked="" type="checkbox"/> vehicle speed	0	0.10%	0	0

## Post-Processing Simulation Output

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

Write a custom criteria script in the Test Manager **Custom Criteria** section of the test case. Custom criteria results appear in the **Results and Artifacts** pane. Results are shown for individual MATLAB Unit Test qualifications. For more information, see “Apply Custom Criteria to Test Cases” on page 6-111.

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

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

## Run-Time Assessments

### verify Statements

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

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

For information on using `verify` statements in your model, see “Activate `verify` Statements” on page 3-14. For information on `verify` statement syntax, see “Run-Time Assessments” on page 3-49.

### assert Statements

Similar to `verify` statements, you can use `assert` statements in a Test Assessment or Test Sequence block. `assert` statements stop simulation, so consider `assert` statements for conditions that could render a test invalid. Failures are shown in the Diagnostic Viewer. `assert` statement results do not appear in the Test Manager results or the Simulation Data Inspector. For more information, see “Activate `verify` Statements” on page 3-14 and “Run-Time Assessments” on page 3-49.

### Run-Time Assessments in Real-Time Tests

If you are using a real-time test case, or if you want to reuse a desktop simulation test case on a real-time target, use `verify` statements instead of `assert` statements. `assert` statements are not supported for real-time testing.

### Model Verification Library Blocks

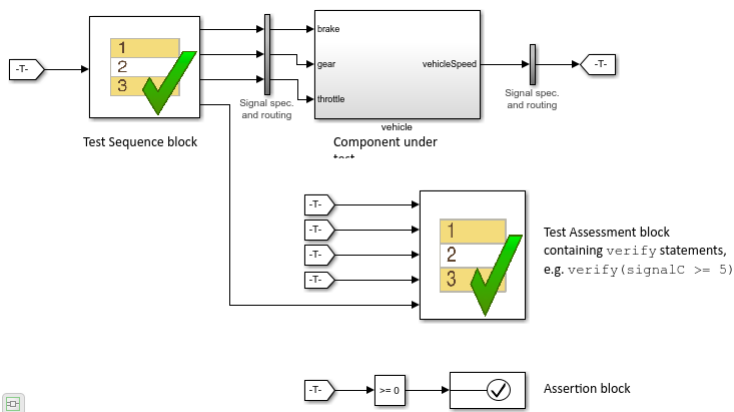
Use blocks from the Model Verification library to assess signals in your model or test harness. For example, you can check that a signal stays between two dynamic bounds, or check that each data point in a signal stays outside of a fixed range. You can view `pass`,

fail, or untested results in the Simulation Data Inspector. For more information, see “View Graphical Results From Model Verification Library” on page 3-84.

## Examples

For example, this test harness includes:

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



## See Also

### Related Examples

- “Test Model Output Against a Baseline” on page 6-9
- “Test Two Simulations for Equivalence”
- “Run-Time Assessments” on page 3-49

# Activate verify Statements

This topic describes how to activate `verify` statements based on simulation conditions. When you use `verify` statements in your model, you must define:

- 1 When each statement is active, and
- 2 The condition that each statement verifies.

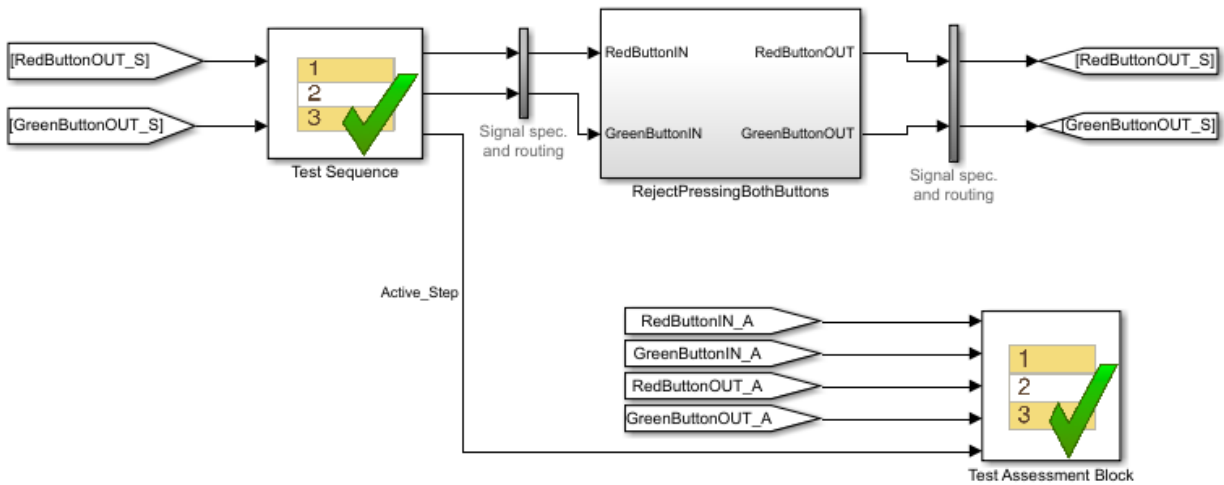
Ensure that each `verify` statement is active at the intended simulation time, and that signals assessed by the `verify` statement are not delayed. Follow these guidelines:

- Author `verify` statements in a separate Test Assessment block. You can create a Test Assessment block during harness creation, or add it from the Simulink Test block library.
- If each step in your Test Sequence block source corresponds with a condition to verify, activate each `verify` statement with a test sequence step. See “Verify Based on the Active Test Step” on page 3-14.
- If your model does not use a Test Sequence block source, activate each `verify` statement with a signal condition. See “Verify Using Signal Conditions” on page 3-17.
- To verify multiple conditions in a single time step, include `verify` statements inside `if` blocks, and include multiple `if` blocks in a single test step. See “Verify Multiple Conditions at a Time” on page 3-18.

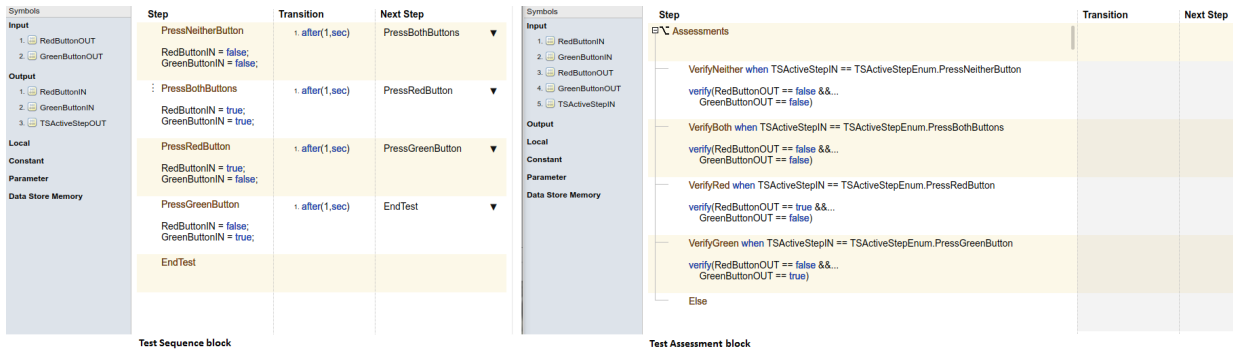
## Verify Based on the Active Test Step

If your model or test harness uses a Test Sequence block source, activate each `verify` statement in the Test Assessment block with a Test Sequence step. Activating `verify` statements with the test step creates a clear relationship between model conditions and verifications.

For example, this test harness shows a Test Sequence and Test Assessment block connected by the `Active_Step` signal.



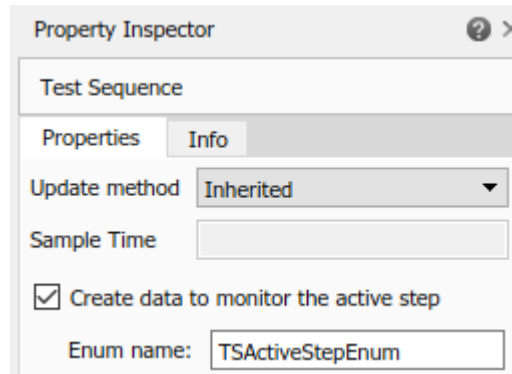
The Test Assessment block contains verify statements in a When decomposition sequence. Each verify statement corresponds to a Test Sequence block step. For more information, see “When Decomposition Transitions” on page 3-24.



To connect the Test Assessment and Test Sequence blocks with active step data:

- 1 Create active step data output for the Test Sequence block:
  - a Select the Test Sequence block in the block diagram.
  - b Create a new enumerated data output. In the Property Inspector, select **Create data to monitor the active step**.

- c Name the enumeration.



- 2 Create a data input for the Test Assessment block:
  - a Open the Test Assessment block.
  - b In the **Symbols** sidebar, next to **Input**, click the **Add data** icon. Name the input.
- 3 In the block diagram, connect the Test Sequence block output to the Test Assessment block input.
- 4 Create a When decomposition sequence in the Test Assessment block.
  - a The Test Assessment block contains a default When decomposition sequence. To change from a standard sequence to a When decomposition sequence, right-click the parent step and select **When decomposition**.
  - b For each substep in the When decomposition sequence, enter a statement that defines when the step is active. Use the active step enumeration data in the when statement. For example, the **VerifyBoth** Test Assessment step is active when the active Test Sequence step is **PressBothButtons**:  

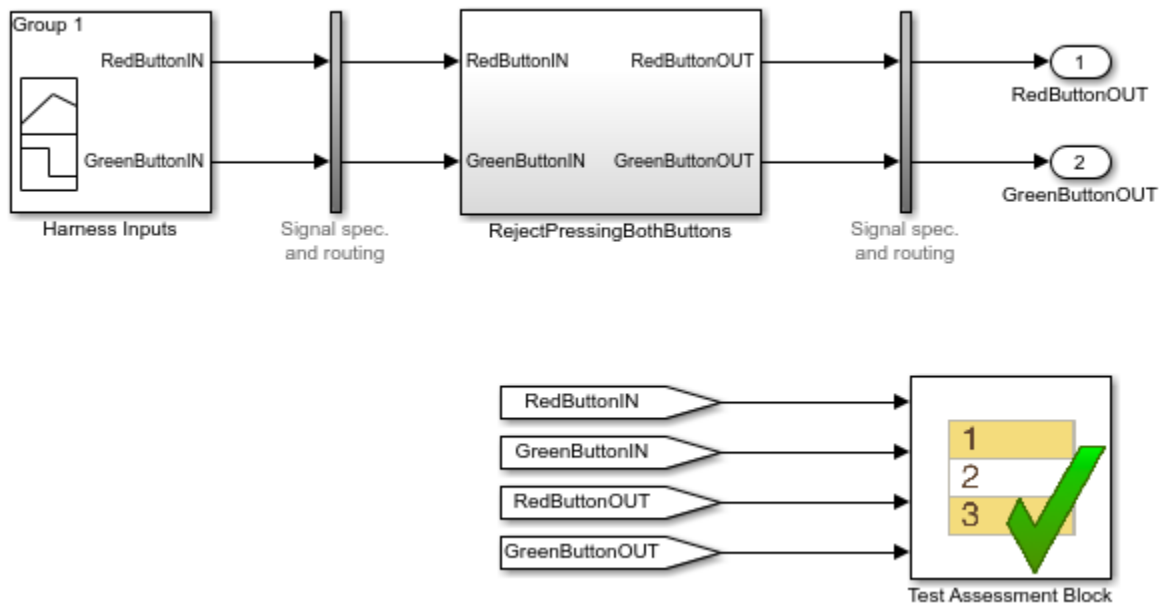
```
VerifyBoth when TSActiveStepIN == TSActiveStepEnum.PressBothButtons
```
  - c Add verify statements to each assessment step.
- 5 You can view the step activity when you run the model. This graphic shows the animation in both blocks. For more information, see “Debug a Test Sequence” on page 3-69.



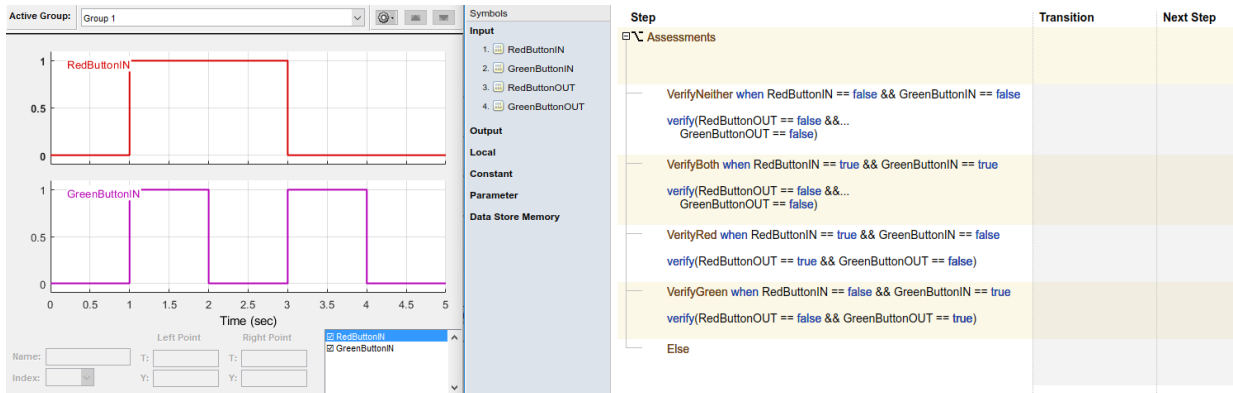
## Verify Using Signal Conditions

If your model does not use a Test Sequence block source, or if you cannot map Test Sequence block steps directly to Test Assessment block steps, use unique signal conditions to activate `verify` statements. Use a separate Test Assessment block, and place `verify` statements in a `When` decomposition sequence. In the `when` conditions, use unique combinations of signals or other data from the model. For more information, see “When Decomposition Transitions” on page 3-24.

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



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



## Verify Multiple Conditions at a Time

To verify multiple conditions that can be valid in a single time step, include verify statements inside if blocks, and include multiple if blocks in a single test step. The test step is active during simulation, and the if blocks are evaluated at each time step.

For example, suppose that you must assess these conditions:

### Parallel Input Conditions and Expected Outputs

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

More than one of these input conditions can be valid at the same time step. To assess these conditions, this Test Assessment block includes six verify statements in the first test step, contained in if blocks:

Symbols	Step	Transition	Next Step
<b>Input</b> 1. RedButtonIN 2. GreenButtonIN 3. RedButtonOUT 4. GreenButtonOUT  <b>Output</b>  <b>Local</b>  <b>Constant</b>  <b>Parameter</b>  <b>Data Store Memory</b>	<b>Assessments</b>  <pre> if RedButtonIN == false &amp;&amp; GreenButtonIN == false   verify(RedButtonOUT == false &amp;&amp; GreenButtonOUT == false) end  if GreenButtonIN == false   verify(GreenButtonOUT ~= true) end  if RedButtonIN == false   verify(RedButtonOUT ~= true) end  if RedButtonIN == true &amp;&amp; GreenButtonIN == true   verify(RedButtonOUT == false &amp;&amp; GreenButtonOUT == false) end  if RedButtonIN == true &amp;&amp; GreenButtonIN == false   verify(RedButtonOUT == true &amp;&amp; GreenButtonOUT == false) end  if RedButtonIN == false &amp;&amp; GreenButtonIN == true   verify(RedButtonOUT == false &amp;&amp; GreenButtonOUT == true) end </pre>		

## See Also

“Test Sequence Editor” on page 3-20 | Test Sequence | Test Assessment


## Related Examples

- “Requirements-Based Testing for Model Development”

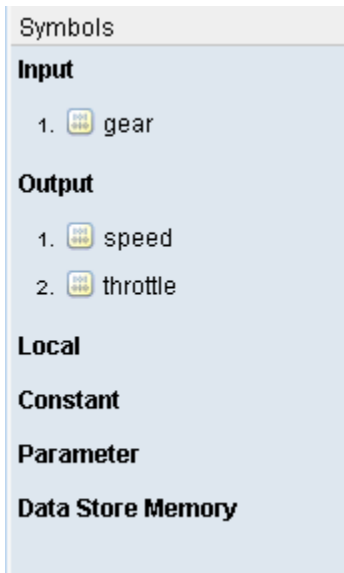
## Test Sequence Editor

### Input, Output, and Data Management

Manage inputs, outputs, and data objects using the **Symbols** sidebar of the Test

Sequence Editor. Click the symbols sidebar button  on the toolbar to show or hide the sidebar. To add a symbol, hover over the symbol type and click **Add**. To edit or delete a data symbol, hover over the data symbol and click **Edit** or **Delete**.

If you add a symbol to the test sequence block, you can access that symbol from test steps at any hierarchy level. For information on using messages, see “Test Step Actions and Transitions” on page 3-32.



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

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

## Add and Delete Test Steps

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

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

## Copy Test Steps

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

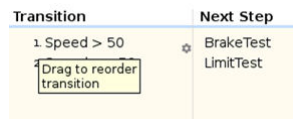
You can also use **Ctrl+X**, **Ctrl+C**, and **Ctrl+V** shortcuts.

## Reorder Test Steps and Transitions

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



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



## Change Test Step Hierarchy

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

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

## Standard Test Step Transitions

In a standard transition, the default step is the first step listed in the sequence. The sequence progresses according to the transition conditions and next steps. To create a sequence using standard transitions:

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

```
on_off = false;  
verify(FanOn == true);
```

- For each step that requires a transition, hover over the **Transition** cell and click **Add transition**. Define the step exit conditions in the transition. For example, this code transitions to another step after the current step has been active for 20 seconds.

```
after(20, sec)
```


- Select the next test step from the drop-down list in the **Next Step** cell.

Step	Transition	Next Step
initialize on_off = false; Tproj = single(0);	1. true	Normal_on_off ▼
□ Normal_on_off end_test = 0;		
On on_off = true;	1. FanOn == true	Wait ▼
Wait on_off = false; verify(FanOn == true,... 'Simulink:verify_scenario1',... 'Fan should be active');	1. after(20,sec)	Off ▼
Off on_off = true;	1. FanOn == false	End ▼
End on_off = false; end_test = 1;		

## When Decomposition Transitions

In a When decomposition sequence, steps activate based on a signal condition that you define on the same line as the test step name. Precede the signal condition with the when operator. For example, a step named OverSpeed2 activates when the signal gear is equal to 2:

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

A When decomposition requires a parent step. To specify a When decomposition sequence, right-click the parent step and select **When decomposition**. The parent step displays the icon . Add substeps to define the when conditions.

At each time step, the when statements evaluate from top to bottom, and the first step with a valid statement executes. The final step in a When decomposition handles



conditions which do not match another when condition in the sequence, similar to an Else condition in an if-then-else construct. Therefore, the final step cannot include a when statement, but the final step can include step actions.

Step	Transition	Next Step
<div style="border: 1px solid #ccc; padding: 5px;"> <div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> <span style="font-size: 1.2em;">▣</span> <span style="font-size: 1.2em;">\</span> </div> <div> <b>HighLevelAssessment</b>   <code>assert(speed &gt;= 0);</code>  <code>assert(throttle &gt;= 0);</code>  <code>assert(throttle &lt;= 100);</code>  <code>assert(gear &gt; 0);</code> </div> </div> </div>		
<div style="border-left: 1px solid #ccc; border-right: 1px solid #ccc; padding: 5px;"> <div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> <span style="font-size: 1.2em;"> </span> </div> <div> <b>OverSpeed3</b> when gear == 3  <code>verify(speed &lt;= 90, 'SLTest:Gear3_overspeed')</code> </div> </div> </div>		
<div style="border-left: 1px solid #ccc; border-right: 1px solid #ccc; padding: 5px;"> <div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> <span style="font-size: 1.2em;"> </span> </div> <div> <b>OverSpeed2</b> when gear == 2  <code>verify(speed &lt;= 50, 'SLTest:Gear2_overspeed')</code> </div> </div> </div>		
<div style="border-left: 1px solid #ccc; border-right: 1px solid #ccc; padding: 5px;"> <div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> <span style="font-size: 1.2em;"> </span> </div> <div> <b>OverSpeed5</b> when gear == 3  <code>verify(speed &lt;= 30, 'SLTest:Gear1_overspeed')</code> </div> </div> </div>		
<div style="border-left: 1px solid #ccc; border-right: 1px solid #ccc; padding: 5px;"> <div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> <span style="font-size: 1.2em;"> </span> </div> <div> <b>Else</b> </div> </div> </div>		

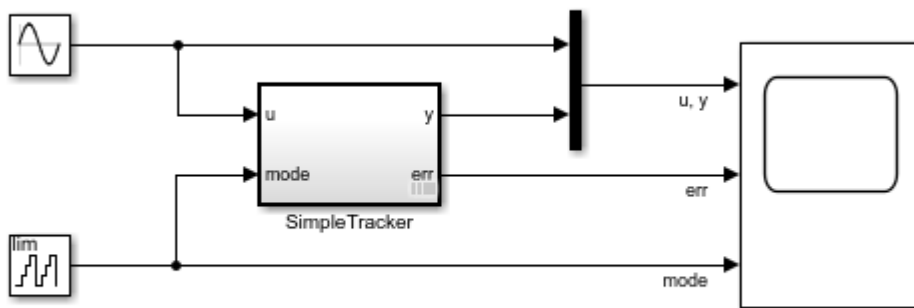
### Using When Decomposition to Write Tests

Assess a model using a When decomposition sequence.

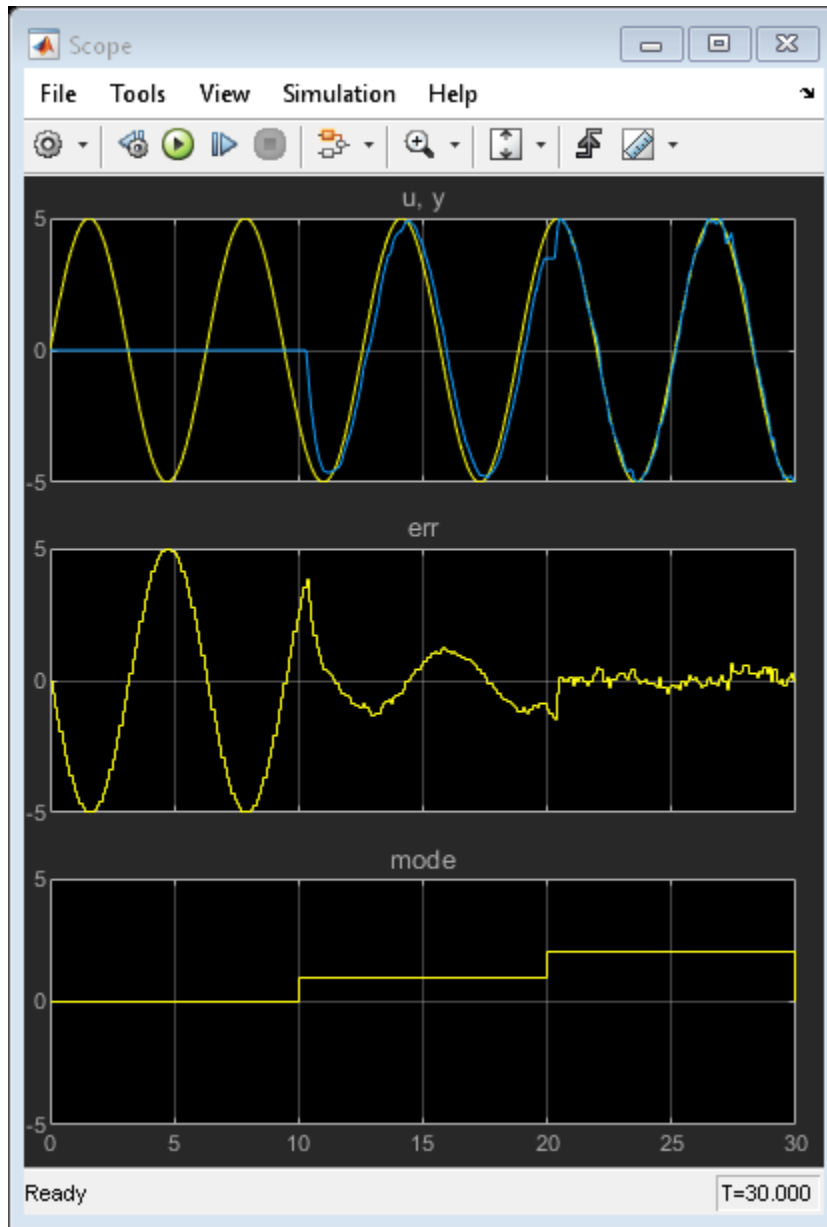
This example shows how to use When decomposition in a Test Sequence block to author assessments in a test harness. The example model implements a simple signal tracker that operates in three modes: off (0), slow (1) and quick (2). Simulate the model and observe the output and error of the signal tracker.

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

### Using When Decomposition to Write Tests



Copyright 2015 The MathWorks, Inc.

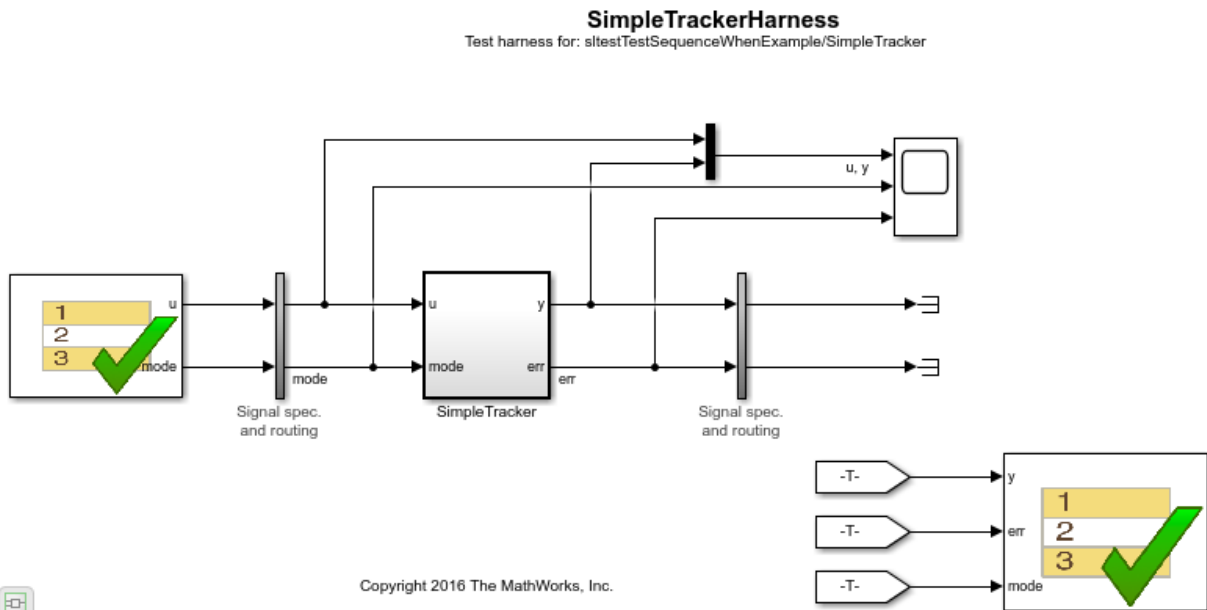


### 3 Test Sequences and Assessments

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

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

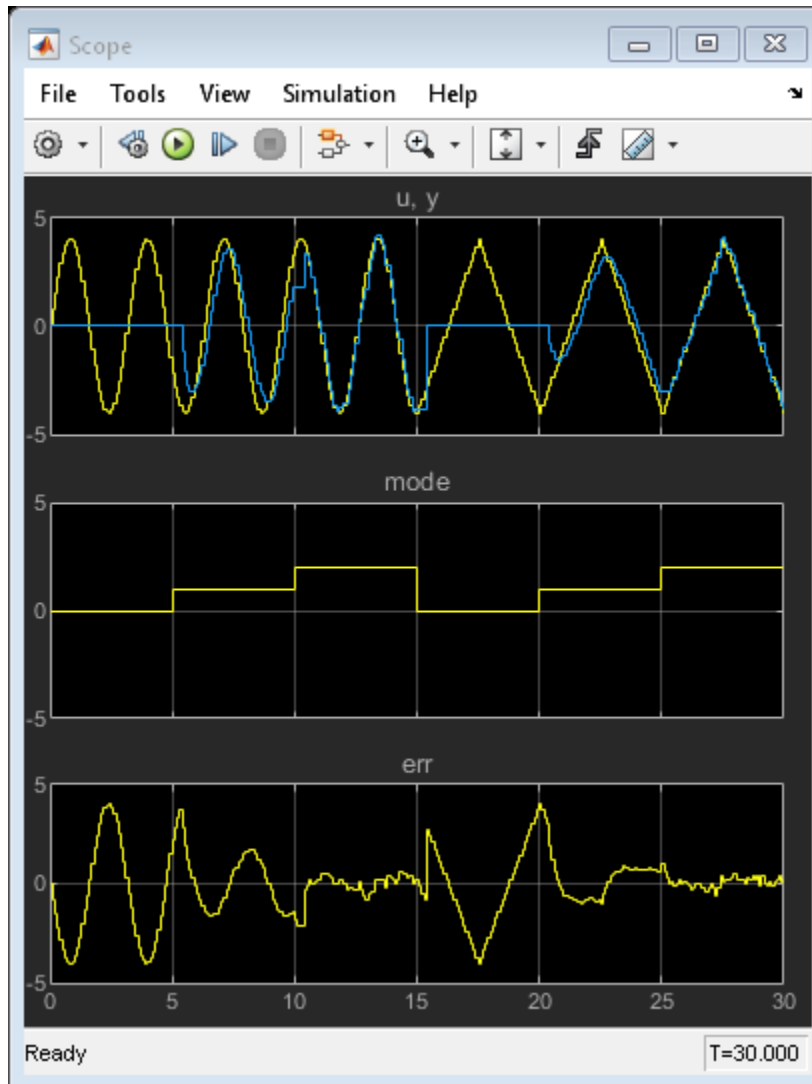
```
open_system mdl;
sltest.harness.open([mdl '/SimpleTracker'], 'SimpleTrackerHarness');
open_system('SimpleTrackerHarness/Test Assessment');
```



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

Simulate the test harness to run the assessments.

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



Close the test harness and main model.

```
close_system mdl, 0;
```

```
clear mdl;
```

## See Also

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

## Related Examples

- “Programmatically Create a Test Sequence” on page 3-56

## Test Step Actions and Transitions

### In this section...

“Transition Between Steps Using Temporal or Signal Conditions” on page 3-32

“Temporal Operators” on page 3-33

“Transition Operators” on page 3-34

“Use Messages in Test Sequences” on page 3-35

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



## Temporal Operators

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

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

Syntax in the table uses these arguments:

#### **TimeUnits**

The units of time

Value: `sec` | `msec` | `usec`

Examples:

`msec`

#### **Condition**

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

Examples:

`u > 0`

`x <= 1.56`

## **Transition Operators**

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

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

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

## Use Messages in Test Sequences

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



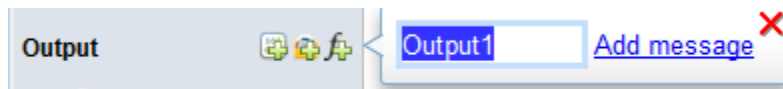
`receive(M)` determines whether a message is present in the input queue M, and removes the message from the queue. `receive(M)` returns `true` if a message is in the queue, and

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

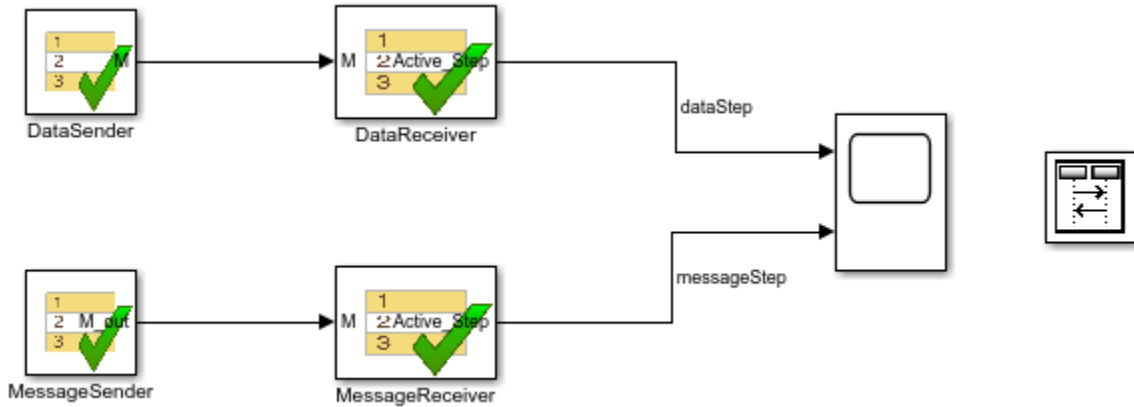
This example demonstrates message inputs and outputs, sending, and receiving a message. The model compares two pairs of test sequences. Each pair is comprised of a sending and receiving test sequence block. The first pair sends and receives data, and the second sends and receives a message.

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			

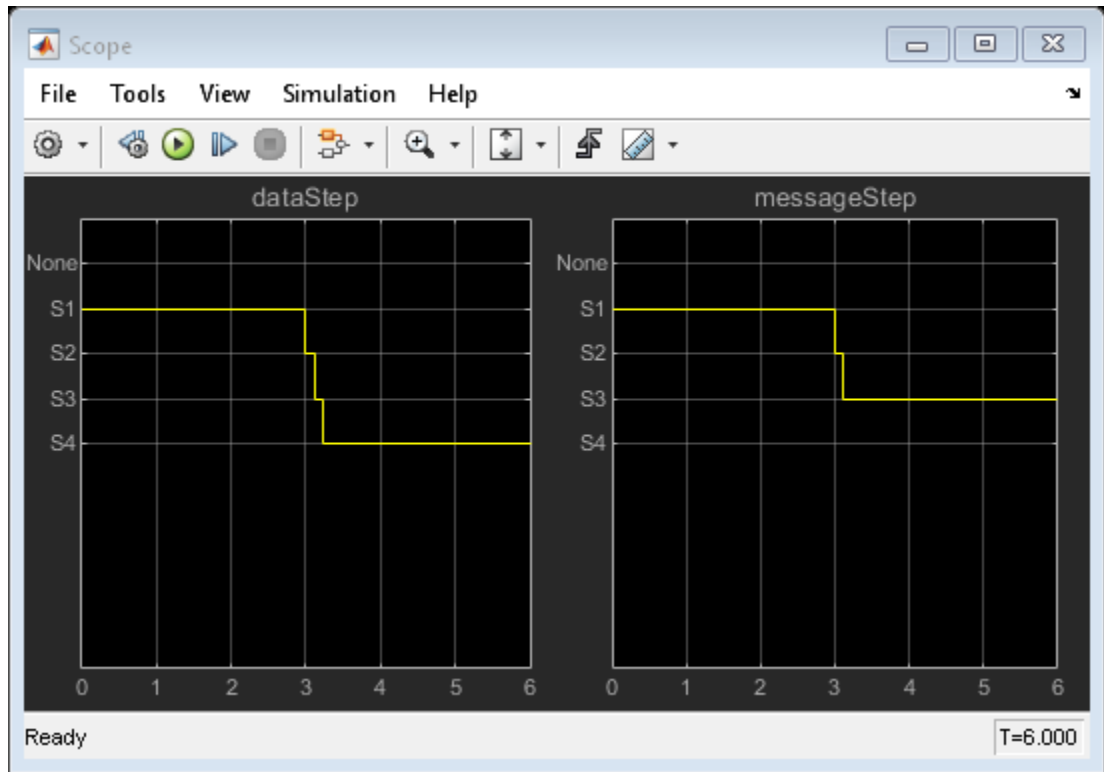
The MessageReceiver block waits 3 seconds, then transitions to step S2. Step S2's transition evaluates the queue `M` with `receive(M)`, removing the message from the queue. `receive(M)` returns `true` since the message is present. `M.data == 3.5` compares the message data to the expected value. The statement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



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

## See Also

“Syntax for Test Sequences and Assessments” on page 3-60 | Test Sequence

## Related Examples

- “Signal Generation Functions” on page 3-41
- “Run-Time Assessments” on page 3-49



## Signal Generation Functions

### In this section...

“Sinusoidal and Random Number Functions in Test Sequences” on page 3-41

“Using an External Function from a Test Sequence Block” on page 3-43

“Signal Generation Functions” on page 3-45

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


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

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

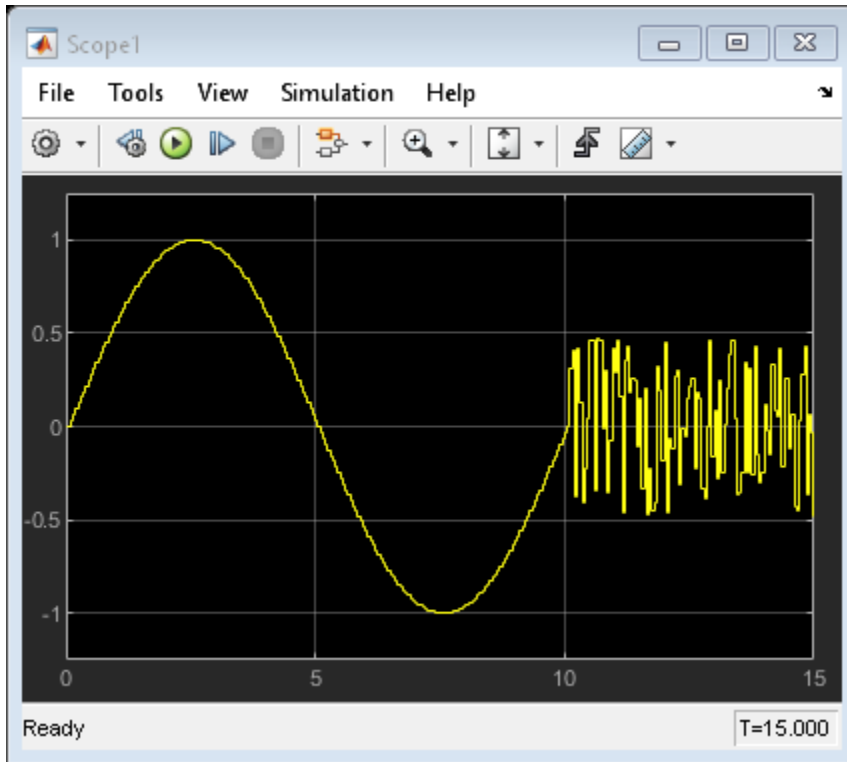
### Sinusoidal and Random Number Functions in Test Sequences

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

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

Symbols	Step	Transition	Next Step
<b>Input</b>	<b>Initialize</b> sg = 0;	1. true	Sine ▼
<b>Output</b> 1.  sg	<b>Sine</b> sg = sin(et*2*pi/10);	1. after(10,sec)	Stop ▼
<b>Local</b> nr	<b>Stop</b> sg = 0;	1. true	Random ▼
<b>Constant</b>	<b>Random</b> coder.extrinsic('rand'); nr = rand; sg = nr - 0.5;	1. after(10,sec)	End ▼
<b>Parameter</b>	<b>End</b> sg = 0;		
<b>Data Store Memory</b>			

The test sequence produces signal sg.



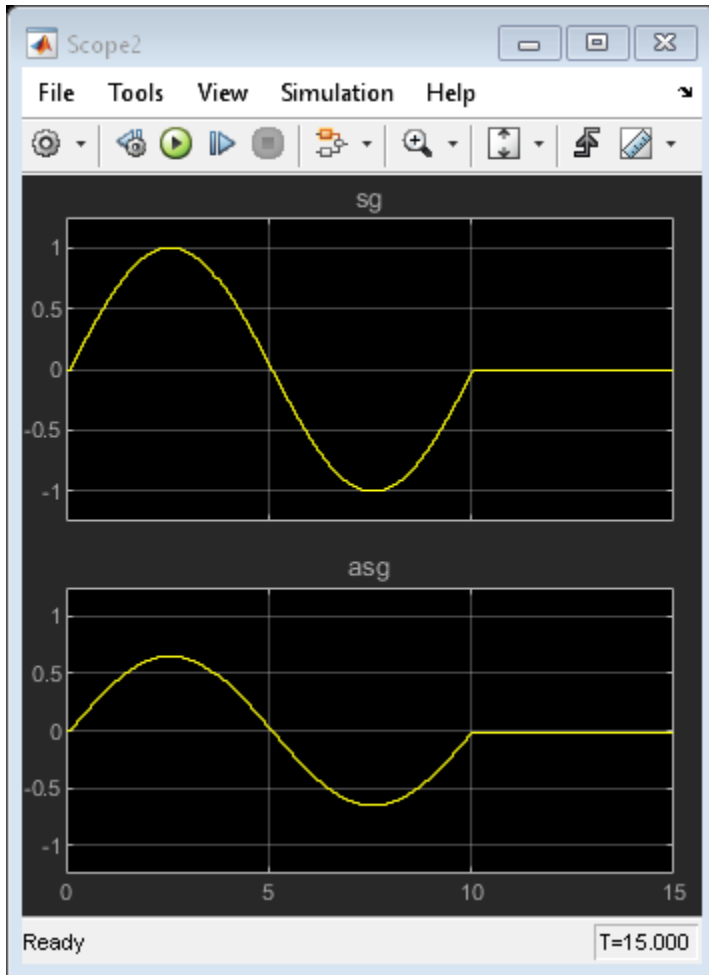
## Using an External Function from a Test Sequence Block

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

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

Symbols	Step	Transition	Next Step
<b>Input</b>	<b>Initialize</b> sg = 0;	1. true	ReducedSine ▼
<b>Output</b>	<b>ReducedSine</b> sg = $\sin(e^t \cdot 2 \cdot \pi / 10)$ ; asg = Attenuate(sg);	1. after(10,sec)	Stop ▼
1. sg 2. asg			
<b>Local</b>			
<b>Constant</b>			
<b>Parameter</b>	<b>Stop</b> sg = 0;		
<b>Data Store Memory</b>			

The test sequence produces signal sg and attenuated signal asg.



## Signal Generation Functions

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

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

Function	Syntax	Description	Example
<code>latch</code>	<code>latch(x)</code>	Saves the value of <code>x</code> at the first time <code>latch(x)</code> evaluates in a test step, and subsequently returns the saved value of <code>x</code> . Resets the saved value of <code>x</code> when the step exits. Reevaluates <code>latch(x)</code> when the step is next active.	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 <code>x</code> , where <code>x</code> 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 cosine of <code>x</code> , where <code>x</code> is in radians.	A cosine wave with a period of 10 sec:  <code>cos(et*2*pi/10)</code>
<code>rand</code>	<code>rand</code>	Uniformly distributed pseudorandom values	Generate new random values for each simulation by declaring <code>rand</code> extrinsic with <code>coder.extrinsic</code> . Assign the random number to a local variable. For example:  <code>coder.extrinsic('rand')</code> <code>nr = rand</code> <code>sg = a + (b-a)*nr</code>

Function	Syntax	Description	Example
randn	randn	Normally distributed pseudorandom values	Generate new random values for each simulation by declaring randn extrinsic with coder.extrinsic. Assign the random number to a local variable. For example:  <code>coder.extrinsic('randn')</code> <code>nr = randn</code> <code>sg = nr*2</code>
exp	exp(x)	Returns the natural exponential function, $e^x$ .	An exponential signal progressing at one tenth of the test step elapsed time:  <code>exp(et/10)</code>

## See Also

“Syntax for Test Sequences and Assessments” on page 3-60 | Test Sequence

## Related Examples

- “Test Step Actions and Transitions” on page 3-32
- “Run-Time Assessments” on page 3-49



## Run-Time Assessments

### In this section...

“verify” on page 3-49

“assert” on page 3-51

“Assessment Statements” on page 3-52

“Logical Operators” on page 3-53

“Relational Operators” on page 3-54

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

### verify

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

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

### Syntax

A `verify` statement uses syntax of these forms

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

The simplest `verify` statement uses only a logical expression. To make results easier to interpret, use additional arguments to define an error message and a statement identifier. Error messages display in the diagnostic viewer. You can use error messages to display key values at the time the statement fails.

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

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

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

```
'SimulinkTest:x_equals_y'
```

#### Continuous-Time Considerations

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

If your model uses continuous time and you use `verify` statements in a Test Sequence or Test Assessment block, consider explicitly setting a discrete block sample time.

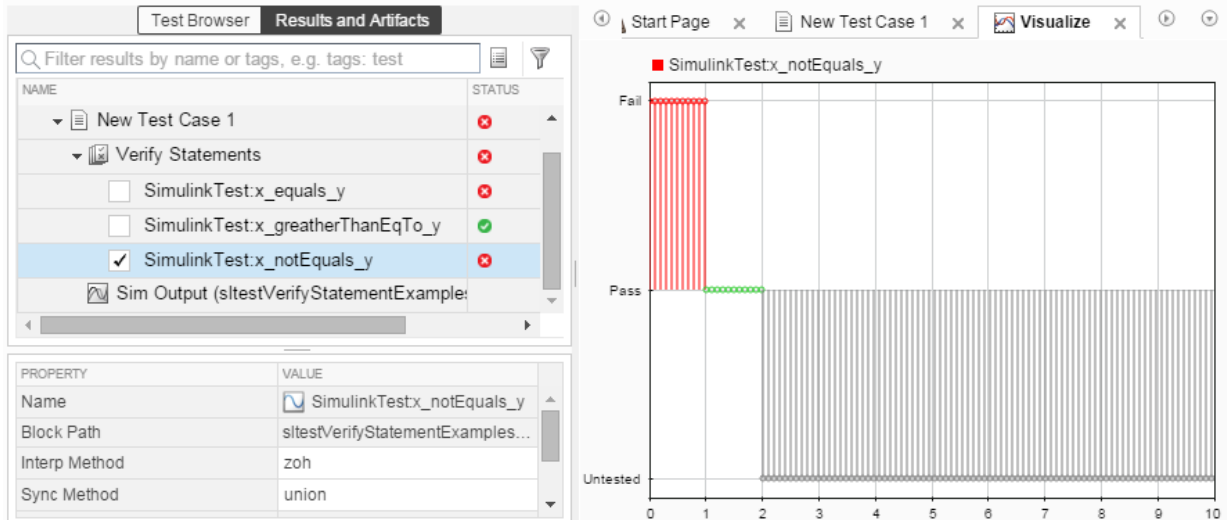
#### Example

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

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

Step	Transition	Next Step
<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_greatherThanEqTo_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  $\theta$ , 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

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

Keyword	Statement Syntax	Description	Example
verify	<pre>verify(expression) verify(expression, errorMessage) verify(expression, identifier, errorMessage)</pre>	Assesses a logical expression. Optional arguments label results in the Test Manager and diagnostic viewer.	<pre>verify(x &gt; y,... 'SimulinkTest:greaterThan',... 'x and y values are %d, %d',x,y)</pre>
assert	<pre>assert(expression) assert(expression, errorMessage)</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>

Syntax in the table uses these arguments:

**expression**

Logical statement assessed

Examples:

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

**identifier**

Label applied to results in the Test Manager

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

Examples:

```
'SimulinkTest:greaterThan'
```

**errorMessage**

Label applied to messages in the diagnostic viewer

Value: String

Examples:

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

**Logical Operators**

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

Operation	Syntax	Description	Example
Negation	<code>~p</code>	not <code>p</code>	<code>verify(~p)</code>
Conjunction	<code>p &amp;&amp; q</code>	<code>p</code> and <code>q</code>	<code>verify(p &amp;&amp; q)</code>
Disjunction	<code>p    q</code>	<code>p</code> or <code>q</code>	<code>verify(p    q)</code>

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

## Relational Operators

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

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

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

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

## See Also

“Syntax for Test Sequences and Assessments” on page 3-60 | Test Sequence

## **Related Examples**

- “Test Step Actions and Transitions” on page 3-32
- “Signal Generation Functions” on page 3-41

## Programmatically Create a Test Sequence

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

### Create a Test Harness Containing a Test Sequence Block

1. Load the model.

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

2. Create the test harness.

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

### Author the Test Sequence

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

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

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

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

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

```
sltest.testsequence.addStepAfter('Harness1/Test Sequence', ...  
    'BrakeTest', 'Initialize1', 'Action', 'endTest = false;')  
  
% Add a transition from |Initialize1| to |BrakeTest|. |  
sltest.testsequence.addTransition('Harness1/Test Sequence', ...
```



```

        'Initialize1', 'true', 'BrakeTest')

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

Open the test harness to view the test sequence.

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

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

The screenshot shows the Test Sequence Editor for 'Harness1/Test Sequence \*'. The interface includes a toolbar with icons for file operations, navigation, and execution. On the left, there are sections for 'Symbols' (Input, Output, Local, Constant, Parameter, Data Store Memory) and a list of symbols. The main area is a table with columns: Step, Transition, Next Step, and Description.

Step	Transition	Next Step	Description
<b>Initialize1</b> %% Initialize data outputs. CruiseOnOff = false; Brake = false; Speed = single(0); CoastSetSw = false; AccelResSw = false;	1. true	BrakeT... ▼	
[ ] <b>BrakeTest</b> endTest = false;	1. endTest == true	Initialize2 ▼	
[ ] <b>SetValue</b> CruiseOnOff = true; Speed = 50;	1. true	Engage ▼	
[ ] <b>Engage</b> CoastSetSw = true;	1. after(2,sec)	Brake ▼	
[ ] <b>Brake</b> CoastSetSw = false; Brake = true;	1. true	Verify ▼	
[ ] <b>Verify</b> verify(engaged == false) endTest = true;			
[ ] <b>Initialize2</b> CruiseOnOff = false; Brake = false; Speed = 0; CoastSetSw = false; AccelResSw = false;	1. true	LimitTest ▼	
[ ] <b>LimitTest</b>			

### Close the Test Harness and Model

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

## Syntax for Test Sequences and Assessments

In this section...
“Assessment Statements” on page 3-60
“Temporal Operators” on page 3-62
“Transition Operators” on page 3-63
“Signal Generation Functions” on page 3-64
“Logical Operators” on page 3-67
“Relational Operators” on page 3-68

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

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

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

```
square(et/10)
```

- Transition when h changes to 0:

```
hasChangedTo(h,0)
```

- Verify that x is greater than y:

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

### Assessment Statements

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

Keyword	Statement Syntax	Description	Example
verify	<pre>verify(expression) verify(expression, errorMessage) verify(expression, identifier, errorMessage)</pre>	Assesses a logical expression. Optional arguments label results in the Test Manager and diagnostic viewer.	<pre>verify(x &gt; y,... 'SimulinkTest:greaterThan',... 'x and y values are %d, %d',x,y)</pre>
assert	<pre>assert(expression) assert(expression, errorMessage)</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>

Syntax in the table uses these arguments:

**expression**

Logical statement assessed

Examples:

`h > 0 && k == 0`

**identifier**

Label applied to results in the Test Manager

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

Examples:

`'SimulinkTest:greaterThan'`

**errorMessage**

Label applied to messages in the diagnostic viewer

Value: String

Examples:

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

**Temporal Operators**

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

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

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

Syntax in the table uses these arguments:

### TimeUnits

The units of time

Value: sec|msec|usec

Examples:

msec

### Condition

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

Examples:

u > 0

x <= 1.56

## Transition Operators

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

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

### Signal Generation Functions

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



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

Function	Syntax	Description	Example
latch	latch(x)	Saves the value of x at the first time latch(x) evaluates in a test step, and subsequently returns the saved value of x. Resets the saved value of x when the step exits. Reevaluates latch(x) when the step is next active.	Latch b to the value of torque:  b = latch(torque)
sin	sin(x)	Returns the sine of x, where x is in radians.	A sine wave with a period of 10 sec:  sin(et*2*pi/10)
cos	cos(x)	Returns the cosine of x, where x is in radians.	A cosine wave with a period of 10 sec:  cos(et*2*pi/10)
rand	rand	Uniformly distributed pseudorandom values	Generate new random values for each simulation by declaring rand extrinsic with coder.extrinsic. Assign the random number to a local variable. For example:  coder.extrinsic('rand') nr = rand sg = a + (b-a)*nr

Function	Syntax	Description	Example
randn	randn	Normally distributed pseudorandom values	Generate new random values for each simulation by declaring randn extrinsic with <code>coder.extrinsic</code> . Assign the random number to a local variable. For example:  <pre>coder.extrinsic('randn') nr = randn sg = nr*2</pre>
exp	exp(x)	Returns the natural exponential function, $e^x$ .	An exponential signal progressing at one tenth of the test step elapsed time:  <pre>exp(et/10)</pre>

## Logical Operators

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

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

## Relational Operators

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

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

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

Operator and Syntax	Description	Example
<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

### Related Examples

- “Run-Time Assessments” on page 3-49
- “Test Step Actions and Transitions” on page 3-32
- “Signal Generation Functions” on page 3-41
- “Programmatically Create a Test Sequence” on page 3-56

## Debug a Test Sequence

### In this section...

“View Test Step Execution During Simulation” on page 3-69

“Set Breakpoints to Enable Debugging” on page 3-69

“View Data Values During Simulation” on page 3-70

“Step Through Simulation” on page 3-71

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.

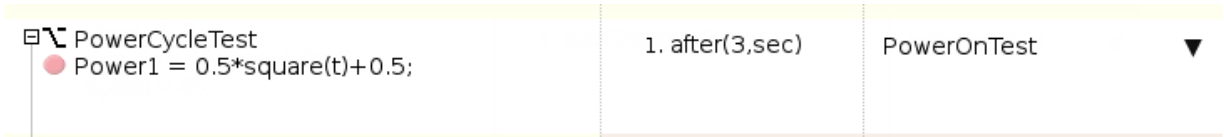
Animation speed affects simulation speed. If you slow down animation speed for debugging, return the speed to **Fast** or **Lightning Fast** when you finish debugging to avoid slowing your simulation. If you do not need the test step highlights and want the fastest simulation, choose **None**.

### 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**.



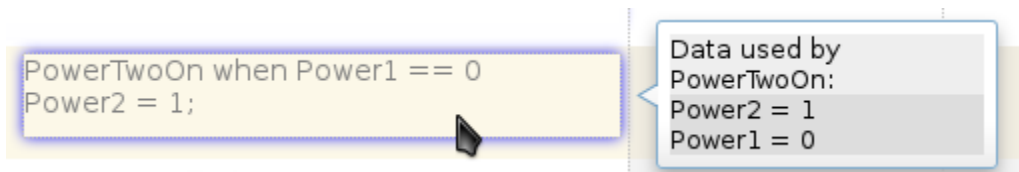
- To add a breakpoint to a test step transition, right-click the test step transition and select **Break when transition taken**.

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 step (for example, using the keyboard shortcuts), the data display does not update. Move off the test step and then hover over the step again to refresh the values.

---

## Step Through Simulation

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

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

## See Also

“Test Sequence Editor” on page 3-20 | Test Sequence

## Test Downshift Points of a Transmission Controller

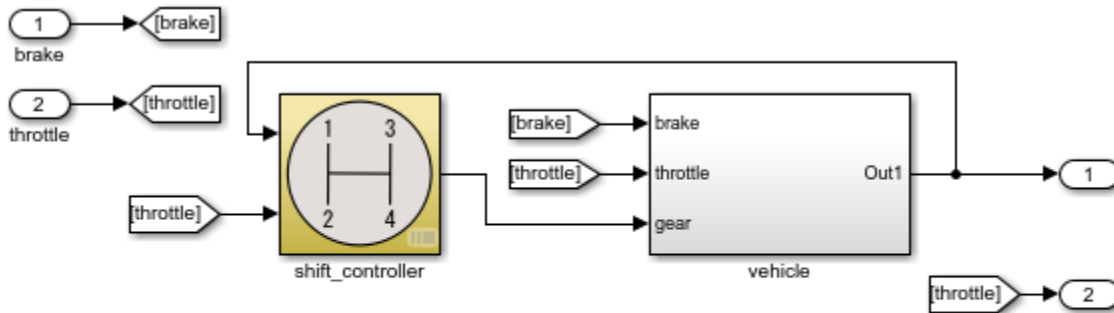
This example demonstrates how to test a transmission shift logic controller using test sequences and test assessments.

### The Model and Controller

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

### The Test

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



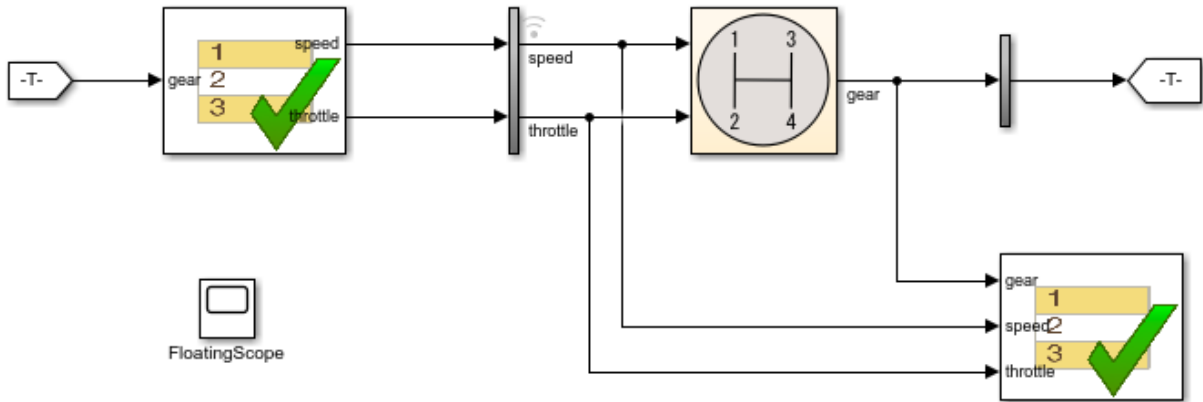
Testing Downshift Points of a Transmission Controller

Copyright 2014-2017 The MathWorks, Inc.

### Open the Test Harness

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





FloatingScope

Copyright 2014-2017 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.

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

This example tests the following conditions:

- Speed shall never be negative.

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

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

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

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

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

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

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

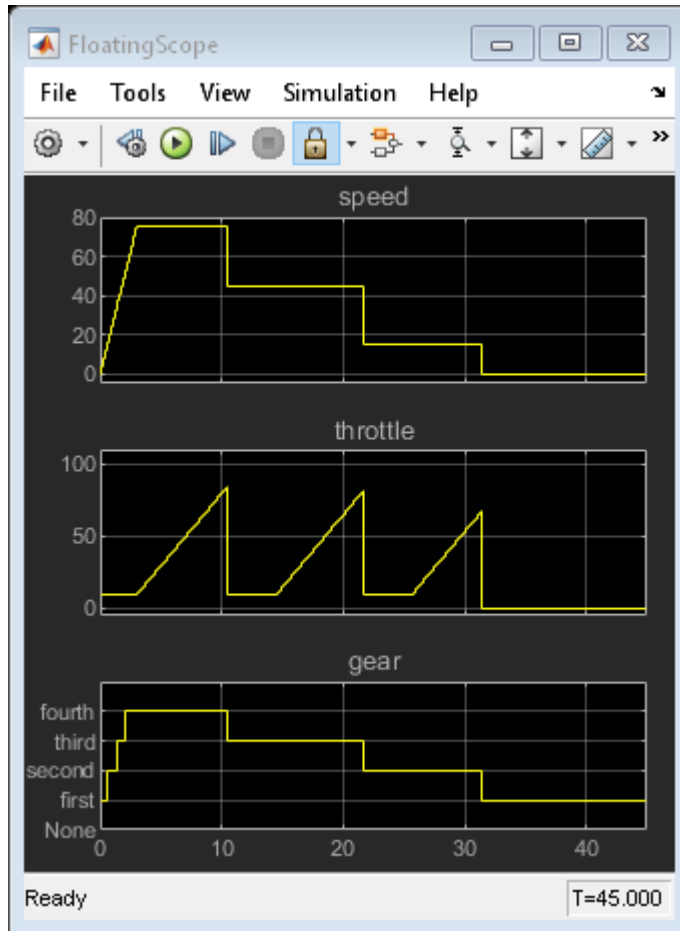
```
OverSpeed2 when gear==2  
verify(speed <= 50, 'Engine overspeed in gear 2')
```

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

Step	Transition	Next Step
<pre> AssertConditions % These conditions ensure simulation validity.  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>		
<pre> OverSpeed3 when gear==3 % Verify speed within specified range for 3rd gear  verify(speed &lt;= 90, 'Engine overspeed in gear 3') </pre>		
<pre> OverSpeed2 when gear==2 % Verify speed within specified range for 2nd gear  verify(speed &lt;= 50, 'Engine overspeed in gear 2') </pre>		
<pre> OverSpeed1 when gear==1 % Verify speed within specified range for 1st gear  verify(speed &lt;= 30, 'Engine overspeed in gear 1') </pre>		
<pre> Else % Else step required for any conditions not corresponding to % the above three when conditions </pre>		

### Testing the Controller

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



### View the Results

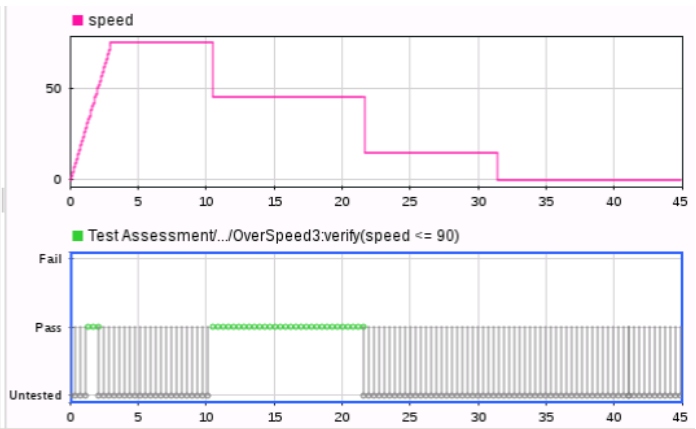
Click the Simulation Data Inspector button in the test harness toolstrip to view the results. You can compare the speed signal to the verify statement outputs.

### 3 Test Sequences and Assessments

Inspect		Compare
Filter Signals		
NAME		LINE
▼ Run 1: controller_harness		
<input type="checkbox"/>	speed	<span style="color: magenta;">█</span>
<input checked="" type="checkbox"/>	Test Assessment/.../OverSpeed3.verify(speed <= 90)	<span style="color: orange;">█</span>
<input type="checkbox"/>	Test Assessment/.../OverSpeed2.verify(speed <= 50)	<span style="color: green;">█</span>
<input type="checkbox"/>	Test Assessment/.../OverSpeed1.verify(speed <= 30)	<span style="color: blue;">█</span>

PROPERTIES	VALUES
Name	Test Assessment/.../OverSpeed3.v...
Line	<span style="color: orange;">█</span>



## Test Assessment Reuse

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

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

### Reuse Test Assessments Using a Library

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

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

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

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

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

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

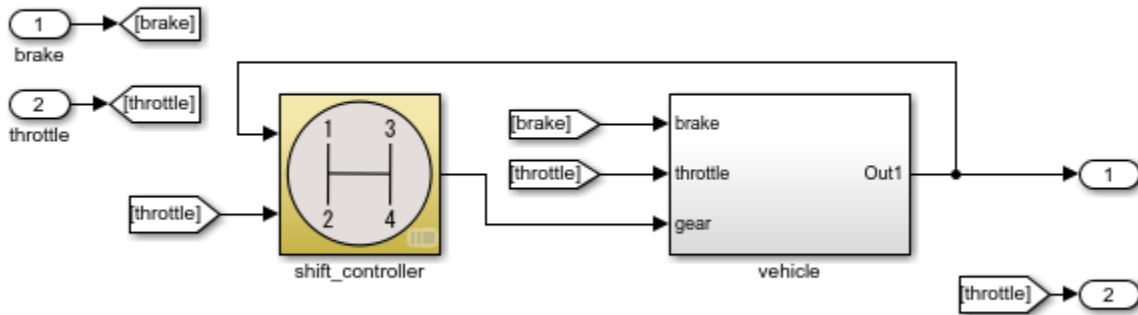
#### Explore the Test Sequence Example Model

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

```
sltestTestSequenceExample
```

#### Testing Downshift Points of a Transmission Controller

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

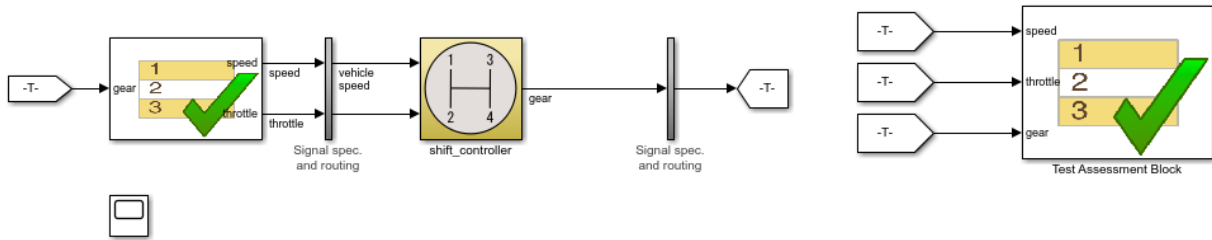


Copyright 2016 The MathWorks, Inc.

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



Test Harness belonging to sitestTestSequenceExample/shift\_controller



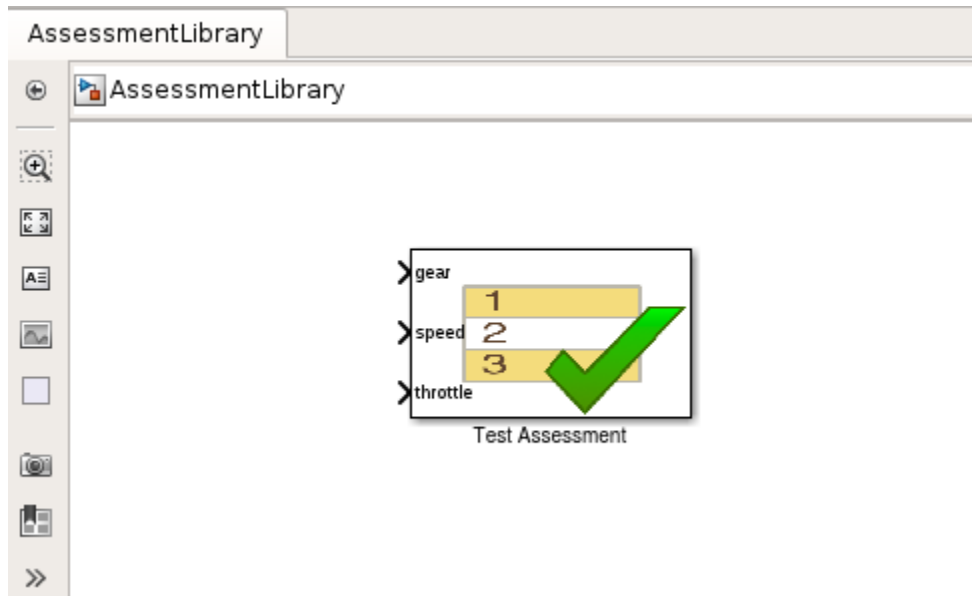
Copyright 2016 The MathWorks, Inc.

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

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

### Create a Library for the Test Assessments

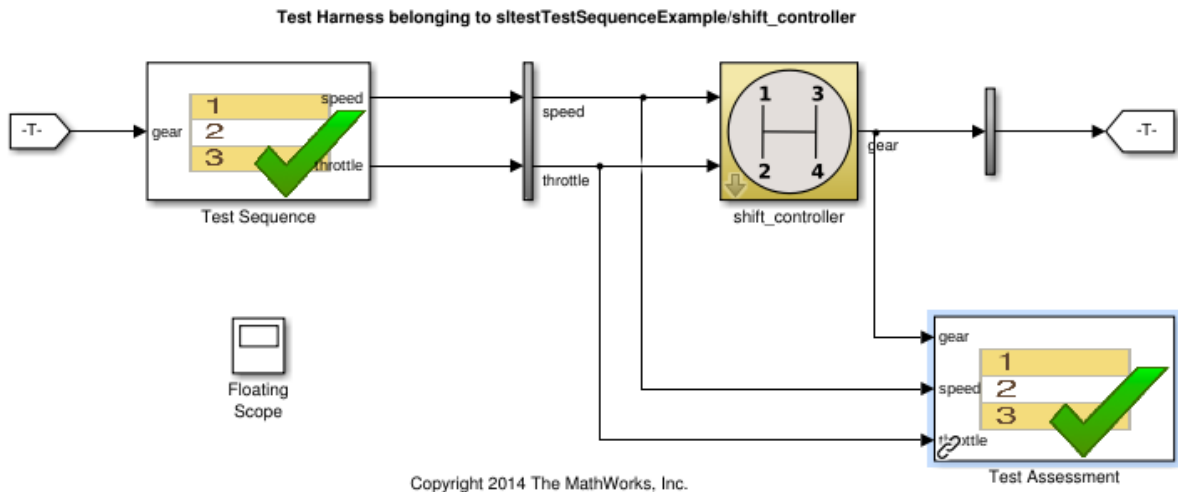
- 1 In the test harness, select **File > New > Library**.
- 2 Save the new library as `AssessmentLibrary` in a writable location on the MATLAB® path.
- 3 Copy the Test Assessment block from the test harness to the library, and then delete the Test Assessment block from the test harness.
- 4 Save the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



#### Create a Linked Test Assessment Block in Test Harnesses

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

- 1 In the test harness, enable the library link display. Select **Display > Library Links > All**.
- 2 Copy the Test Assessment block from `AssessmentLibrary` into `controller_harness`. The block displays a library link badge.
- 3 Connect the signal inputs to the Test Assessment block.



### Edit the Assessment Block in the Library

- 1 Unlock the library. Select **Diagram > Unlock Library**.
- 2 Add a fifth assertion to the Test Sequence block: `assert (gear < 5);`
- 3 Save and close the library. Closing locks the library.

# View Graphical Results From Model Verification Library

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

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

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

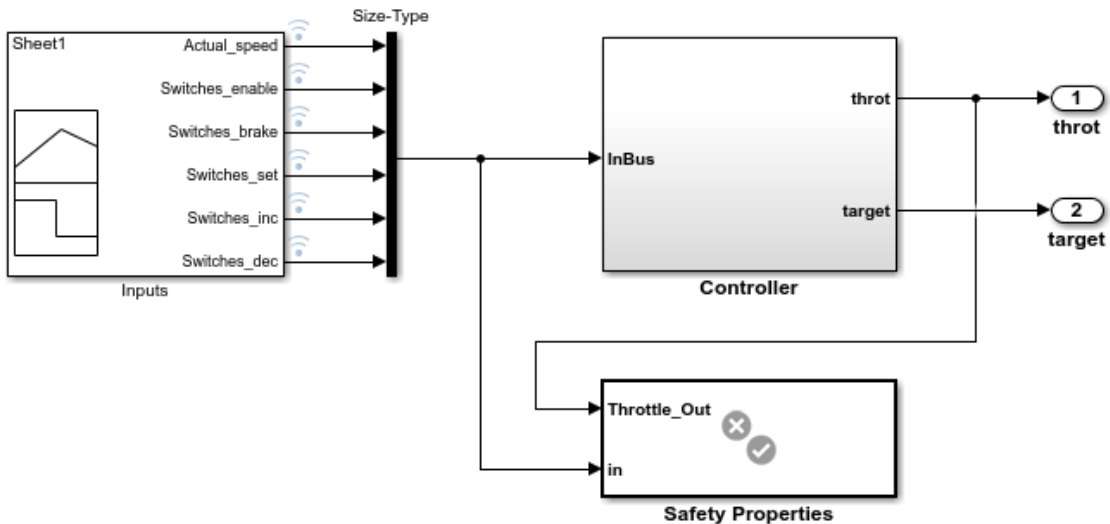
This example shows how to view outputs from Model Verification blocks in the Test Manager or Simulation Data Inspector.

### Open the Model

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

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

## Simulink Test Cruise Control: Output of Model Verification blocks



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

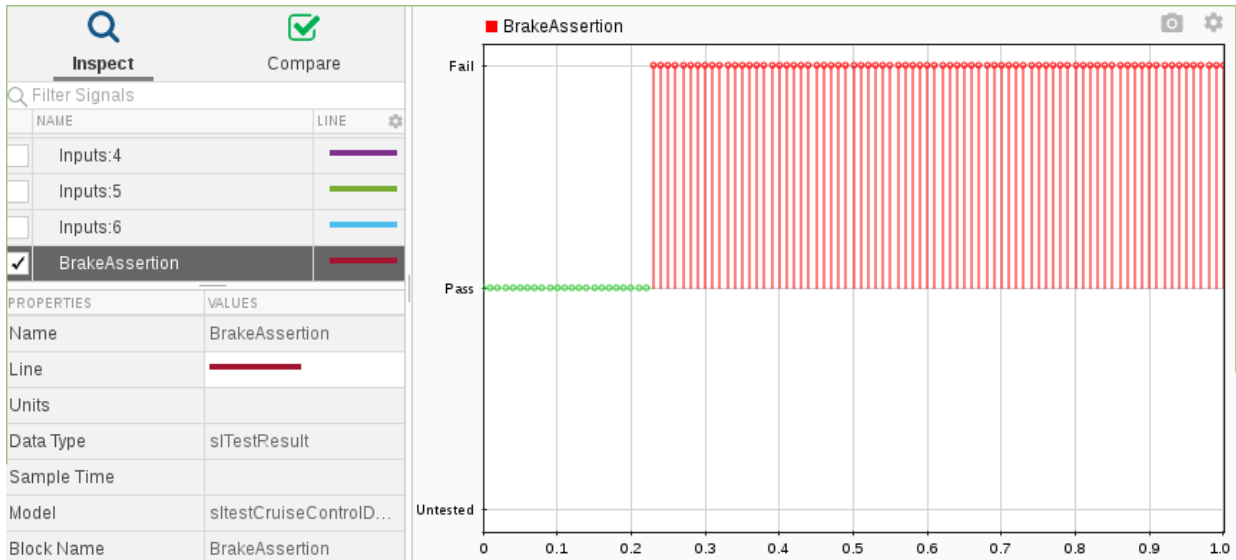
Copyright 2006-2016 The MathWorks, Inc.

### Simulate the Model and View Results in SDI

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

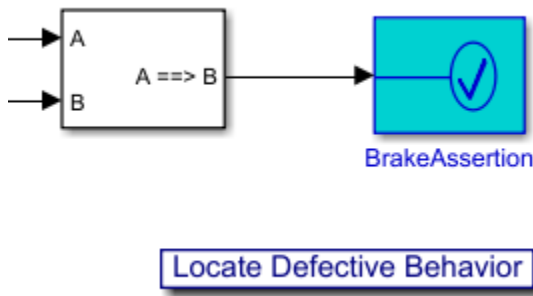
After the simulation completes, open SDI. The results show that the assertion failed at 0.23 seconds.

```
Simulink.sdi.view
```



#### Highlight Assertion Block in the Model

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



#### See Also

`sltest.getAssessments`

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

---

- “SIL Verification for a Subsystem” on page 4-2
- “Test Integrated Code” on page 4-7

# SIL Verification for a Subsystem

<b>In this section...</b>
---------------------------

“Create a SIL Verification Harness for a Controller” on page 4-2
--

“Configure and Simulate a SIL Verification Harness” on page 4-4
---

“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.

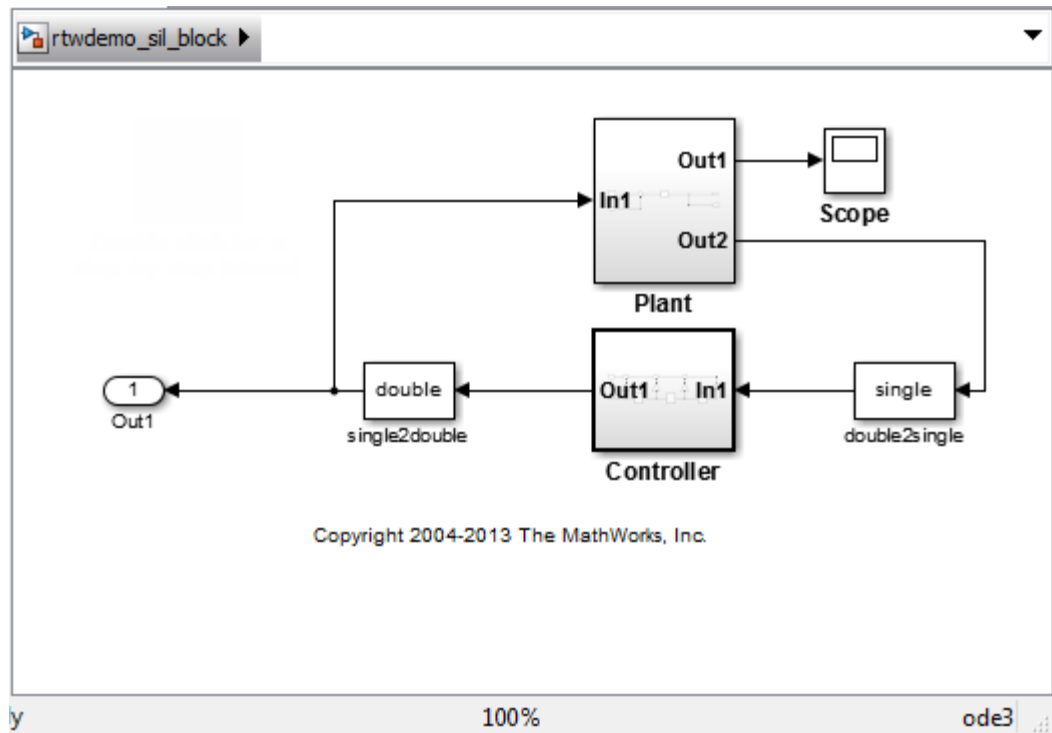
This example models a closed-loop controller-plant system. The controller regulates the plant output.

## Create a SIL Verification Harness for a Controller

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



- 1 Open the example model by entering  
`rtwdemo_sil_block`  
at the MATLAB command prompt,



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

```
set_param(bdroot,'SignalLogging','on','SignalLoggingName',...
'SIL_signals','SignalLoggingSaveFormat','Dataset')
```
- 4 Right-click the signal into Controller port In1, and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `controller_model_input`. Select **Log signal data** and click **OK**.

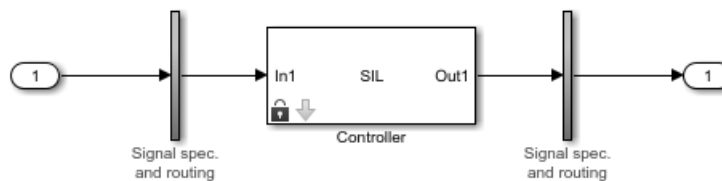
- 5 Right-click the signal out of Controller port Out1, and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `controller_model_output`. Select **Log signal data** and click **OK**.
- 6 Simulate the model.
- 7 Get the logged signals from the simulation output into the workspace. At the command prompt, enter

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

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

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

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



### Configure and Simulate a SIL Verification Harness

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


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

- Data Import/Export** pane, select **Input**. Enter `control_in1.Values` as the input and click **OK**.
- 2 Enable signal logging for the test harness. At the command prompt, enter

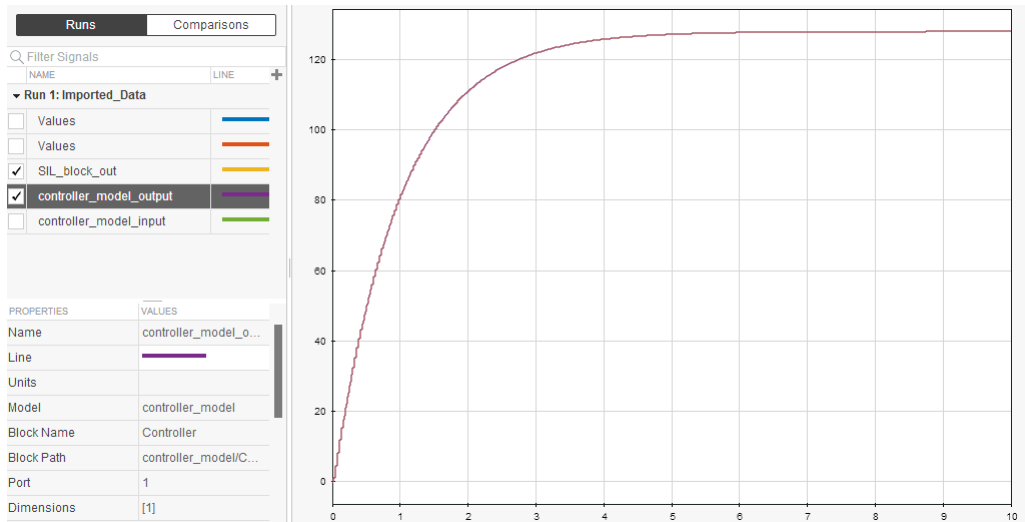
```
set_param('SIL_harness','SignalLogging','on','SignalLoggingName',...
'harness_signals','SignalLoggingSaveFormat','Dataset')
```
  - 3 Right-click the output signal of the SIL block and select **Properties**. In the **Signal Properties** dialog box, for the **Signal name**, enter `SIL_block_out`. Select **Log signal data** and click **OK**.
  - 4 Simulate the harness.


## Compare the SIL Block and Model Controller Outputs

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

- 1 In the test harness model, click the Simulation Data Inspector button  to open the Simulation Data Inspector.
- 2 In the Simulation Data Inspector, click **Import**. In the **Import** dialog box.
  - Set **Import from** to: Base workspace.
  - Set **Import to** to: New Run.
  - Under **Data to import**, select **Signal Name** to import data from all sources.
- 3 Click **Import**.
- 4 Select the `SIL_block_out` and `controller_model_out` signals in the **Runs** pane of the data inspector window.

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



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

## See Also

### More About

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

## Test Integrated Code

### In this section...

“Test Integrated C Code” on page 4-7

“Test Code in S-Functions” on page 4-8

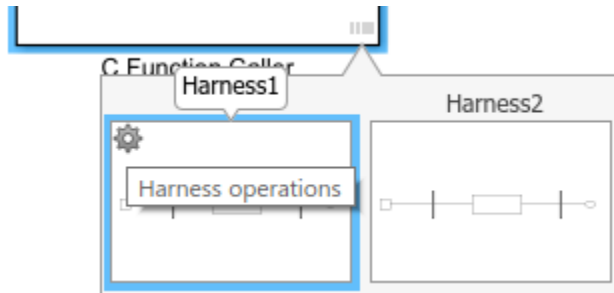
“S-Function Testing Example” on page 4-8

### Test Integrated C Code

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

The C Caller block uses configuration parameters to define the custom code. If you change the configuration parameters, synchronize the parameters between the test harness and the model. For more information, see “Synchronize Changes Between Test Harness and Model” on page 2-57 and “Create Test Harnesses and Select Properties” on page 2-15.

- If you change the test harness configuration parameters, you can push the configuration set to the main model. Select **Analysis > Test Harness > Push Component and Parameters to Main Model**, or use `sltest.harness.push`.
- If you change the main model configuration parameters in the main model, and you want to update the test harness parameters, the test harness must copy the configuration parameters on rebuild. You can set this property in two ways:
  - When you create the test harness, in the **Advanced Properties**, select **Update Configuration Parameters and Model Workspace data on rebuild**.
  - For existing test harnesses, in the harness preview, click the **Harness operations** icon to open the harness properties. In the harness properties, select **Update Configuration Parameters and Model Workspace data on rebuild**.



### Test Code in S-Functions

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

### S-Function Testing Example

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

---

**Note** This example works only on a 64-bit Windows® platform.

---

#### Set Up the Working Environment

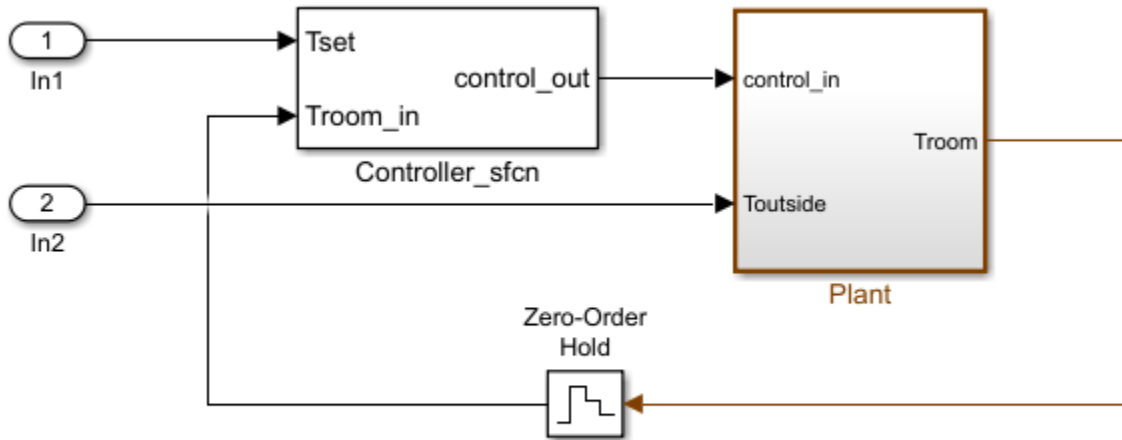
- 1 Add the example folder to the MATLAB path, and set the example file names.

```
ep = fullfile(docroot, 'toolbox', 'sltest', 'examples');  
addpath(ep);
```

```
md = 'sltestHeatpumpSfunExample.slx'  
cb = 'sltestHeatpumpBusPostLoadFcn.mat'  
dt = 'PumpDirection.m'
```

- 2 Open the model.

```
open_system(fullfile(ep,md))
```



Copyright 1990-2018 The MathWorks, Inc.

In the example model:


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

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

Temperature Condition	System State	Fan Command	Pump Command	Pump Direction
$ T_{room\_in} - T_{set}  < \Delta T_{fan}$	idle	0	0	0
$\Delta T_{fan} \leq  T_{room\_in} - T_{set}  < \Delta T_{pump}$	fan only	1	0	0

Temperature Condition	System State	Fan Command	Pump Command	Pump Direction
Troom_in - Tset  >= DeltaT_pump and Tset < Troom_in	cooling	1	1	-1
Troom_in - Tset  >= DeltaT_pump and Tset > Troom_in	heating	1	1	1

### Create a Test Case

- 1 Open the Test Manager by selecting **Analysis > Test Manager** from the Simulink menu.
- 2 From the Test Manager toolstrip, click **New** to create a test file. Name and save the test file.
- 3 In the test case, under **System Under Test**, click the  button to load the current model into the test case.

### Create a Test Harness

- 1 In the model, right-click the Controller\_sfcn subsystem and select **Test Harness > Create for 'Controller\_sfcn'**.
- 2 Set the harness properties.

In the **Basic Properties** tab:

- Set **Name** to test\_harness\_1
  - Set **Sources and Sinks** to **None** and **Scope**
- 3 Click **OK** to create the test harness.
  - 4 In the test case, under **System Under Test**, refresh the test harness list and select test\_harness\_1 for the **Harness**.

### Add Inputs and Set Simulation Parameters

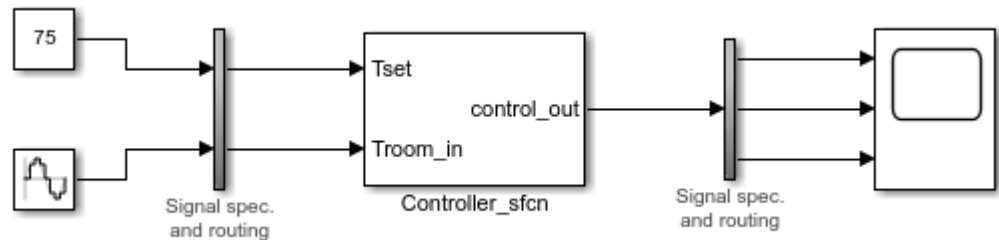
Create inputs in the test harness, with a constant Tset and a time-varying Troom\_in.

- 1 Connect a Constant block to the Tset input and set the value to 75.
- 2 Add a Sine Wave block to the harness model to simulate a temperature signal. Connect the Sine Wave block to the conversion subsystem input Troom\_in.
- 3 Double-click the Sine Wave block and set the parameters:



Parameter	Value
Amplitude	15
Bias	75
Frequency	$2 * \pi / 3600$
Phase (rad)	0
Sample time	1

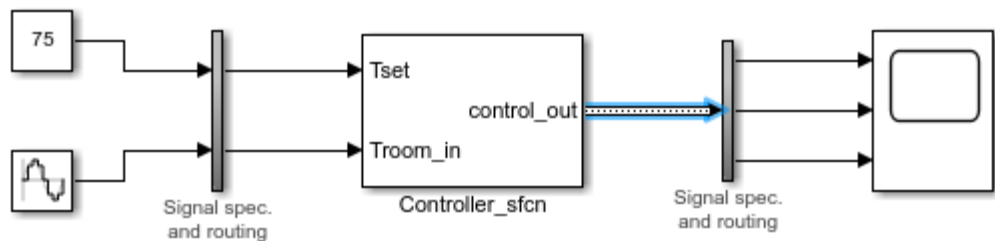
Select **Interpret vector parameters as 1-D**.



- 4 In the **Solver** pane of the Simulink toolstrip, set **Stop time** to 3600.

### Obtain Baseline Data

- 1 In the test case, in **Simulation Outputs**, click **Add**. Highlight the output bus from the controller S-Function.



- 2 In the **Signal Selection** dialog box, click the **Add** button.

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

- Under **Baseline Criteria**, click **Capture** to record a baseline data set from simulating the test harness. Save the baseline data set to the working folder. The baseline signals appear in the table.

▼ SIMULATION OUTPUTS\*

LOGGED SIGNALS

NAME	BLOCK PATH	PORT INDEX
▼ <input checked="" type="checkbox"/> Signal Set 1		
<input checked="" type="checkbox"/> <a href="#">Controller_sfcn:1</a>	test_harness_1/Controller_sfcn	1

Plot signals on the specified plots after simulation + Add ▼ Delete

OTHER OUTPUTS

Override model settings

States  Output

Final states  Data stores  Signal logging

► CONFIGURATION SETTINGS OVERRIDES

▼ BASELINE CRITERIA\*

Include baseline data in test result

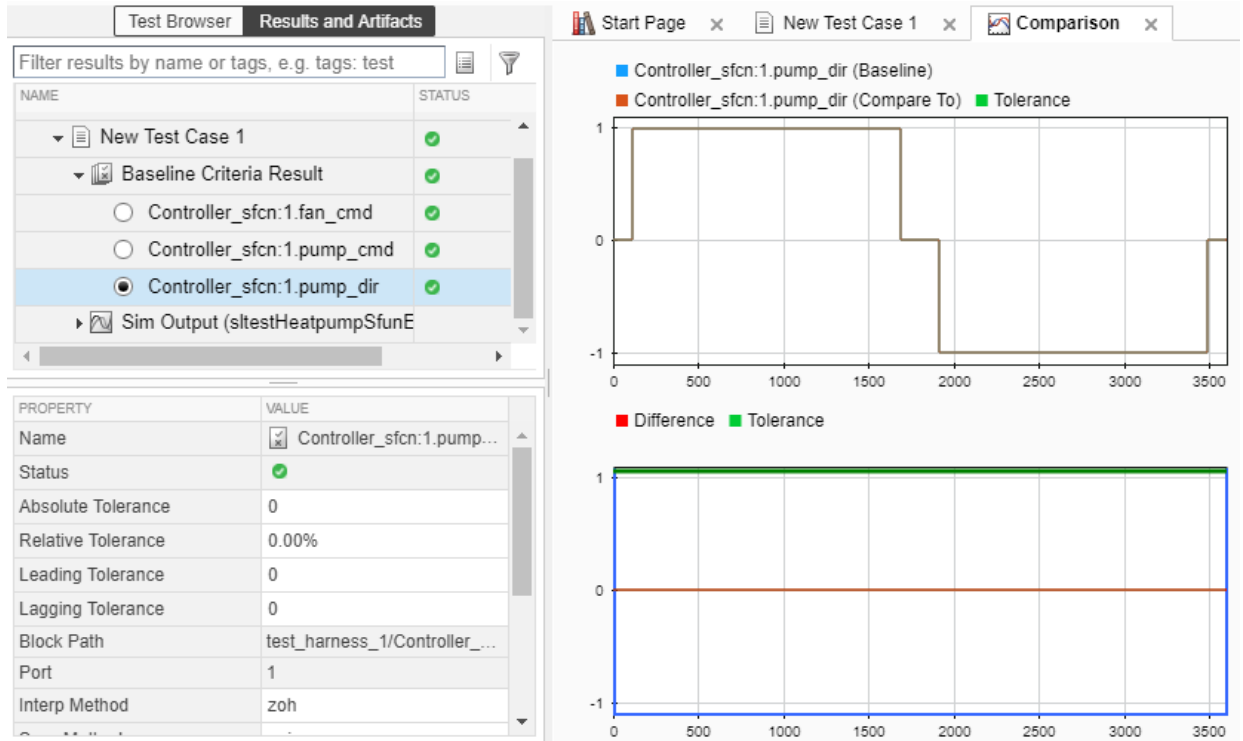
SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL	
▼ <input checked="" type="checkbox"/> sfcn_baseline.mat	0	0.00%	0	0	+
<input checked="" type="checkbox"/> fan_cmd	0	0.00%	0	0	
<input checked="" type="checkbox"/> pump_cmd	0	0.00%	0	0	
<input checked="" type="checkbox"/> pump_dir	0	0.00%	0	0	

+ Add... ↩ Capture... ✎ Edit... ↻ Refresh 📄 Visualize 🗑 Delete

### Run the Test Case and View Results

- Run the test case. The test results appear in the **Results and Artifacts** pane.

- Expand the results to view the baseline criteria result. The baseline test passes because the simulation output is identical to the baseline data.





# **Simulink Test Manager Introduction**

---

## Introduction to Test Manager

In this section...
“Start Test Manager” on page 5-2
“Create Tests and Understand the Test Hierarchy” on page 5-2
“View Test Results” on page 5-4
“Share Results” on page 5-4
“Compare Test Files” on page 5-4

Test Manager in Simulink Test helps you to automate Simulink model testing and organize large sets of tests. You perform model tests in Test Manager using test cases in which you specify the criteria that determine a pass-fail outcome. After you run a test, you can view and share the results.

For help on individual test case items, see “Test Sections” on page 6-95.

### Start Test Manager

You can start Test Manager from a model or from the MATLAB command prompt.

- To start Test Manager from a model, select **Analysis > Test Manager**.
- To start Test Manager from the command prompt, enter: `sltestmgr`.

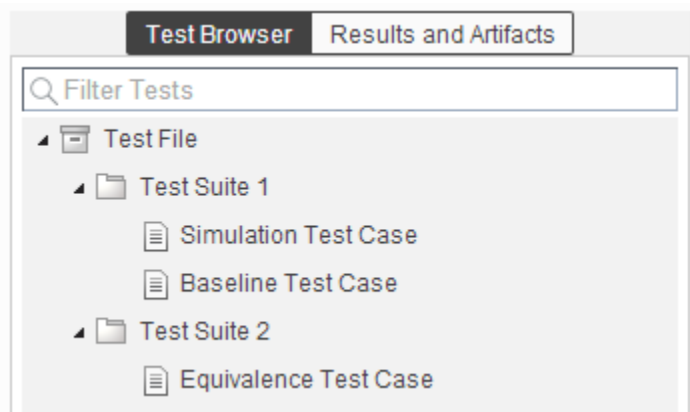
### Create Tests and Understand the Test Hierarchy

In Test Manager, you create test files, which contain one or more test suites that each contain one or more test cases.

To create a test file, select **New > Test File**. Name the file and click **Save**.

The test files and their contents appear in the **Test Browser** pane.

Each new test file contains a test suite, `New Test Suite 1`, which contains a test case, `New Test Case 1`. You can rename test suites and test cases in the browser. The figure shows a test file that contains two test suites that each contain test cases.



Add test suites and test cases to the test file hierarchy using the **New** menu. Use test suites to group related test cases. For each test case, specify details such as the model under test, the simulation outputs to capture, and parameter overrides to apply.

Run tests in the Test Manager, and view results in the **Results and Artifacts** pane. You can run a test file, test suite, or individual test cases.

For baseline and equivalence test cases, you can specify tolerances for the simulation outputs that determine pass or fail. For more information on setting tolerances, see “Apply Tolerances to Test Criteria” on page 6-88.

Using Test Manager, from the **New** menu, you can create these types of test cases:

- **Baseline** — A baseline test is a type comparison test. For a baseline test, you first generate a baseline set of simulation outputs as the basis for comparison. Running a baseline test compares the outputs of the comparison simulation to the baseline. With equivalence tests, you can specify tolerances that determine a range of values that allow the test to pass. That is, the results are equivalent even if not the same. You can set absolute, relative, leading, or lagging tolerances in the **Baseline Criteria** section of the test case. See “Test Model Output Against a Baseline” on page 6-9.
- **Equivalence** — An equivalence test in Test Manager compares signal outputs from two simulations. You can specify tolerances that help the test determine whether the results are equivalent. Set tolerances in the **Equivalence Criteria** section of the test case. See “Test Two Simulations for Equivalence”.
- **Simulation** — A simulation test checks that a simulation runs without errors, including model assertions. See “Test a Simulation for Run-Time Errors” on page 6-13.

- **Real-Time Test** — A baseline, equivalence, or simulation test that runs on the target hardware. See “Test Models in Real Time” on page 8-2.
- **Test Manager Generated Tests** — Tests that you do not need to configure:
  - **Test File from Model**, which generates a test file and uses signal builders and test harnesses in the model as the basis for generating test cases. See “Automatically Create Test Cases for Your Model” on page 6-16.
  - **Test for Subsystem**, which generates a test harness for the subsystem you select and generates a test case to run on the test harness. See “Generate Tests for a Subsystem” on page 6-22.

You can also have Test Manager generate test cases for you based on your model design or for a specific subsystem to test a subsystem in isolation. See “Automatically Create Test Cases for Your Model” on page 6-16 and “Generate Tests for a Subsystem” on page 6-22.

### View Test Results

Tests can pass or fail. If all the criteria defined in a test case are satisfied, within the defined tolerances, then a test passes. If any of the criteria are not satisfied, then the test fails. After the test runs, you can see the results in the **Results and Artifacts** pane. Each test result has a summary page that highlights the outcome of the test: passed, failed, or incomplete. You can also see the simulation output in the results. You can further inspect the signal data from the simulation output using the data inspector view. To view a result in the data inspector view, select it.

### 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 view later in Test Manager, then you can export the results to a file. To archive the results in a document, generate a report, which can include the test outcome, test summary, and criteria used for test comparisons. See “Export Test Results and Generate Reports” on page 7-9.

### Compare Test Files

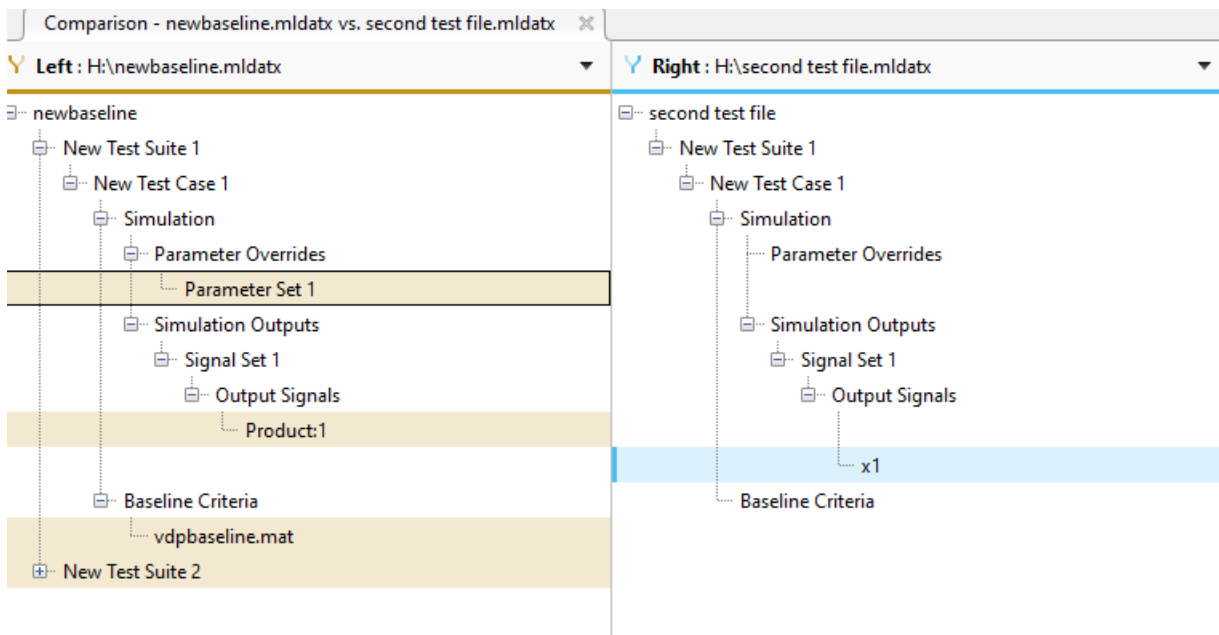
You can use the **Compare** command in the **File** section of the MATLAB toolstrip to compare two test files. Comparing test files is useful for determining the differences



between two similar test files. For example, you can see whether they contain the same test cases and whether those test cases are configured identically.

- 1 From the **File** section of the MATLAB toolstrip, click **Compare**.
- 2 In the **First file or folder** box, enter the first test file that you want to compare. Test files are in the `.mldatx` format.
- 3 In the **Second file or folder** box, enter the second test file that you want to compare.
- 4 For **Comparison type**, select Simulink Test File Comparison. Then click **Compare**.

The figure shows an example of a comparison between two test files. The highlights indicate where one file specifies information that the comparison file does not. For example, `newbaseline.mldatx` includes a test suite that the other file does not contain.



## **See Also**

### **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
- “Automatically Create Test Cases for Your Model” on page 6-16
- “Generate Tests for a Subsystem” on page 6-22
- “Synchronize Test Cases to Model Changes” on page 6-24
- “Specify Microsoft Excel File Format for Signal Data” on page 6-25
- “Author a Test in Excel” on page 6-34
- “Create Test with Data from Excel” on page 6-39
- “Use External Inputs in Test Cases” on page 6-42
- “Create Data Files to Use as Test Inputs” on page 6-49
- “Run Tests in Multiple Releases” on page 6-52
- “Examine Test Failures and Modify Baselines” on page 6-57
- “Automate Tests Programmatically” on page 6-63
- “Run Combinations of Tests Using Iterations” on page 6-70
- “Collect Coverage in Tests” on page 6-81
- “Run Tests Using Parallel Execution” on page 6-86
- “Apply Tolerances to Test Criteria” on page 6-88
- “Test Manager Limitations” on page 6-93
- “Test Sections” on page 6-95
- “Test Models Using Inputs Generated by Simulink Design Verifier” on page 6-106
- “Apply Custom Criteria to Test Cases” on page 6-111
- “Create, Store, and Open MATLAB Figures” on page 6-122
- “Test Models Using MATLAB Unit Test” on page 6-125
- “Tests for Continuous Integration” on page 6-129
- “Filter Test Execution and Results” on page 6-134

## Manage Test File Dependencies

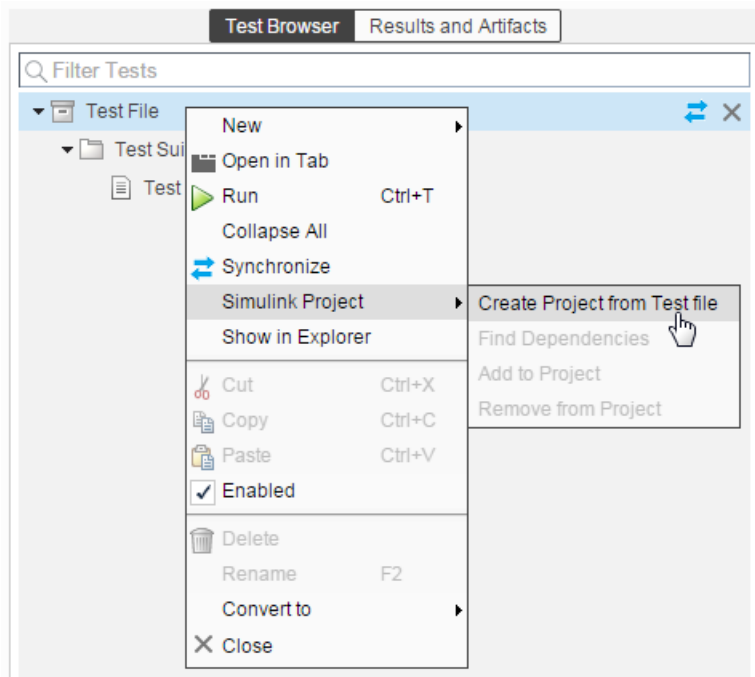
In this section...
“Package a Test File Using Simulink Projects” on page 6-2
“Find Test File Dependencies and Impact” on page 6-4
“Share a Test File with Dependencies” on page 6-8

You can help track and manage your test file dependencies by creating a Simulink Project for your test file and the files it depends on. Examples of test file dependencies include requirements, data files, callbacks, test harnesses, and custom criteria scripts. Packaging test file dependencies in a Simulink Project also helps you share tests with other users.

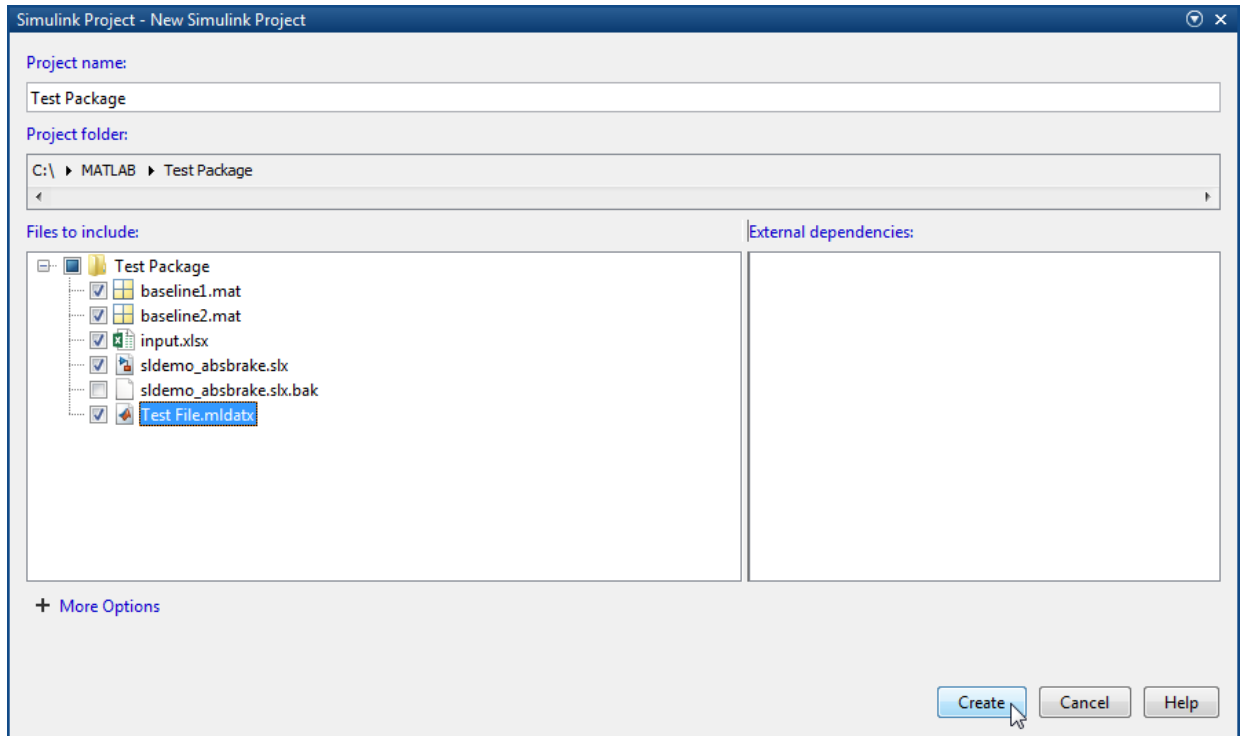
### Package a Test File Using Simulink Projects

- 1 In the **Test Browser**, right-click the test file.
- 2 Select **Simulink Project > Create Project from Test File**.

Simulink Projects open 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.



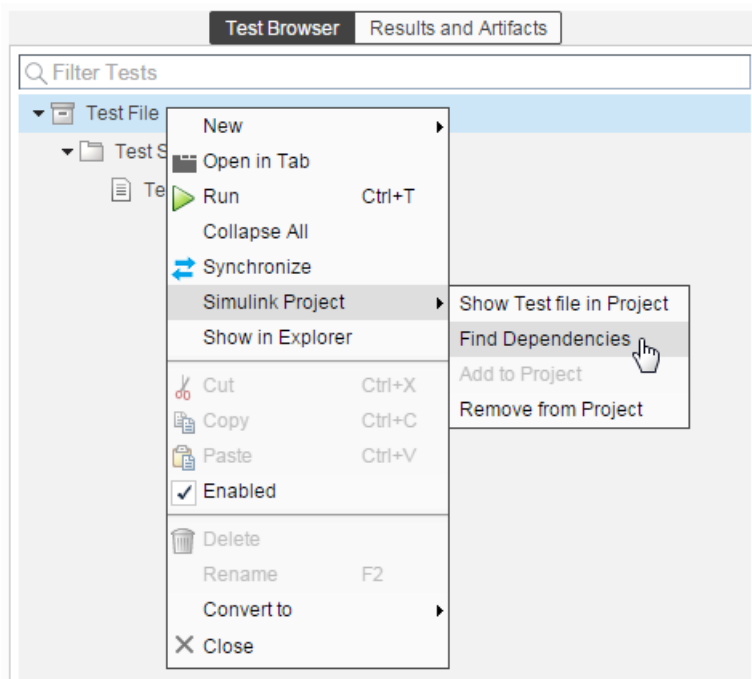
- 3 Specify project name, and verify the list of selected file dependencies.
- 4 Click **Create**.



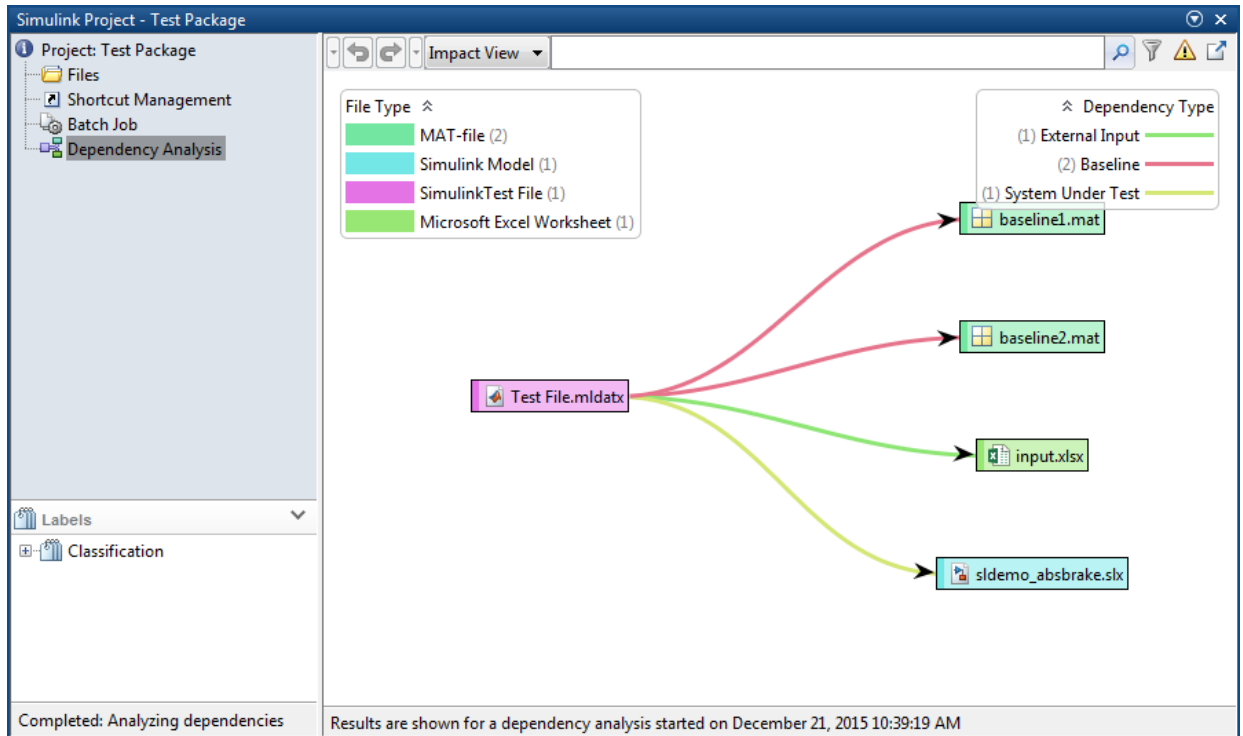
### Find Test File Dependencies and Impact

You can find test file dependencies from the Test Browser. Your test file must be saved in a Simulink project.

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



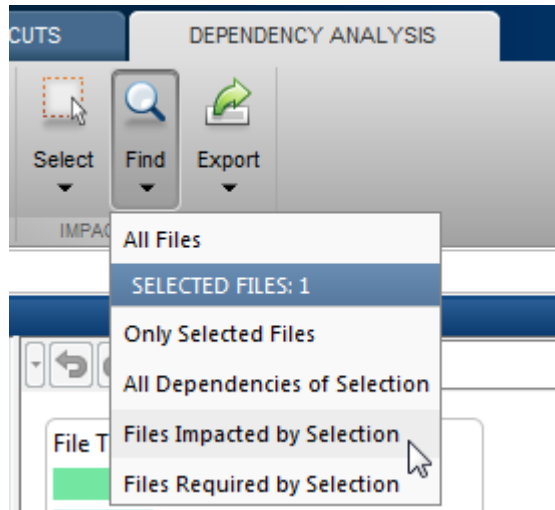
Dependencies are color coded in the file dependency graph.



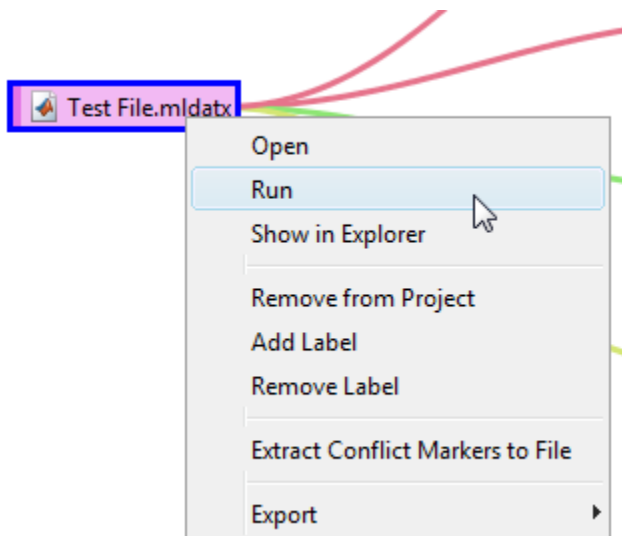
If you want to change a model or requirement, you can determine the potential impact of the change on your tests.

- 1 In the dependency graph, select the item that could impact your tests.
- 2 In the Simulink Projects toolstrip, click **Files > Files Impacted by Selection**.





If you want to run a test file again, then you can right-click the test file in the graph and select **Run**. The Test Manager opens the test file and runs the test cases contained in it.



### **Share a Test File with Dependencies**

You can easily share test files that are already saved in a Simulink Project. If you send the project folder, then it contains the file dependencies for the test file.

### **See Also**

#### **Related Examples**

- [“What Are Simulink Projects?” \(Simulink\)](#)


## Test Model Output Against a Baseline

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


### Create the Test Case

- 1 Open the `sldemo_absbrake` model.
- 2 To open the Test Manager from the model, select **Analysis > Test Manager**.
- 3 From the Test Manager toolstrip, click **New** to create a test file. Name and save the test file.

The test file consists of a test suite that contains one baseline test case. They appear in the **Test Browser** pane.

- 4 Right-click the baseline test case in the **Test Browser** pane, and select **Rename**. Rename the test case to `Slip Baseline Test`.
- 5 Under **System Under Test** in the test case, click the **Use current model** button  to load the `sldemo_absbrake` model into the test case.
- 6 To record a baseline from the system under test, under **Baseline Criteria**, click **Capture**.
- 7 In the Capture Baseline dialog box, for the file format, select Excel. Specify a location to save the baseline to and click **Capture**.
- 8 The baseline criteria file and the logged signals appear in the table. Set the **Absolute Tolerance** of the `Ww` signal to 15.

SIGNAL NAME	SHEETS	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL	
▼ <input checked="" type="checkbox"/> <code>abs_baseline.xlsx</code>	<a href="#">baseline1</a>	0	0.00%	0	0	+
<input checked="" type="checkbox"/> <code>Ww</code>		15	0.00%	0	0	
<input checked="" type="checkbox"/> <code>Vs</code>		0	0.00%	0	0	
<input checked="" type="checkbox"/> <code>Sd</code>		0	0.00%	0	0	
<input checked="" type="checkbox"/> <code>slp</code>		0	0.00%	0	0	

**Tip** To add or remove columns in the baseline criteria table, click the column selector button .

For more information about tolerances and criteria, see “Apply Tolerances to Test Criteria” on page 6-88.

### Run the Test Case and View Results

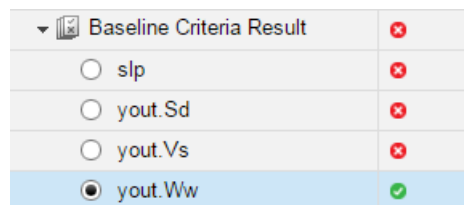
- 1 In the `sldemo_absbrake` model, set the **Desired relative slip** constant block to `0.22`.
- 2 In the Test Manager, select the Slip Baseline Test case in the **Test Browser** pane.
- 3 On the Test Manager toolbar, click **Run**.

In the **Results and Artifacts** pane, the new test result appears at the top of the table.

- 4 Expand the results until you see the baseline criteria result. Right-click the result and select **Expand All Under**.

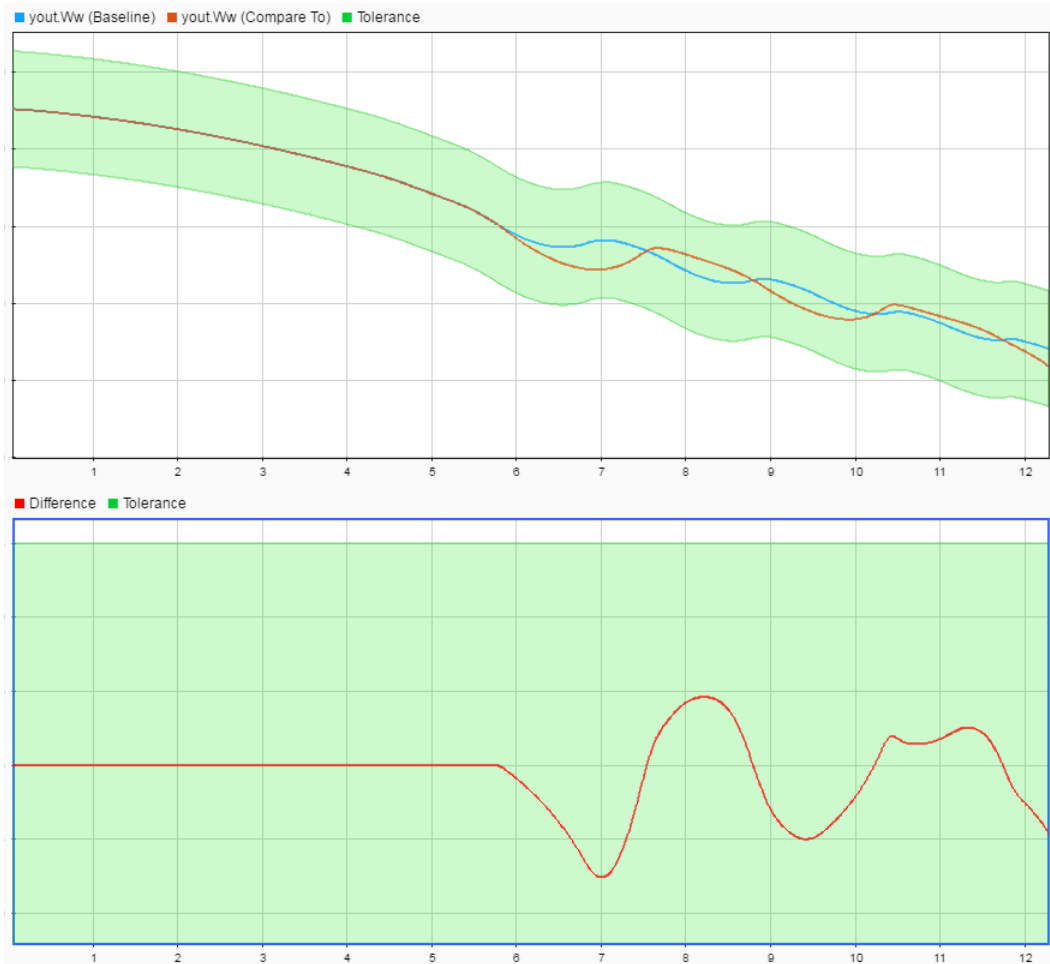
The signal `yout.Ww` passes, but the overall baseline test fails because other signal comparisons specified in the **Baseline Criteria** section of the test case were not satisfied.

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



▼ Baseline Criteria Result	✘
<input type="radio"/> slp	✘
<input type="radio"/> yout.Sd	✘
<input type="radio"/> yout.Vs	✘
<input checked="" type="radio"/> yout.Ww	✔

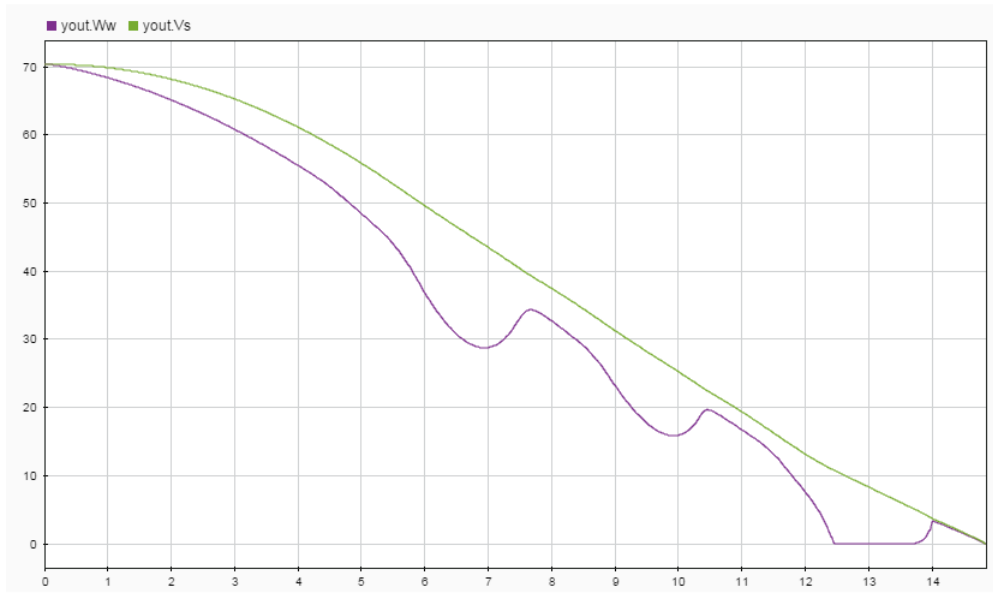
The **Comparison** tab opens and shows the criteria comparisons for the `yout.Ww` signal and the tolerance.



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

▼ <input checked="" type="checkbox"/> Sim Output (sldemo_absbrake)		
<input type="checkbox"/>	slp	—
<input type="checkbox"/>	yout.Sd	—
<input checked="" type="checkbox"/>	yout.Vs	—
<input checked="" type="checkbox"/>	yout.Ww	—

The **Visualize** tab opens and plots the simulation output.



For information on how to export results and generate reports from results, see “Export Test Results and Generate Reports” on page 7-9.

## See Also

### Related Examples

- “Apply Tolerances to Test Criteria” on page 6-88
- “Capture Baseline Criteria” on page 6-102
- “Run Tests in Multiple Releases” on page 6-52

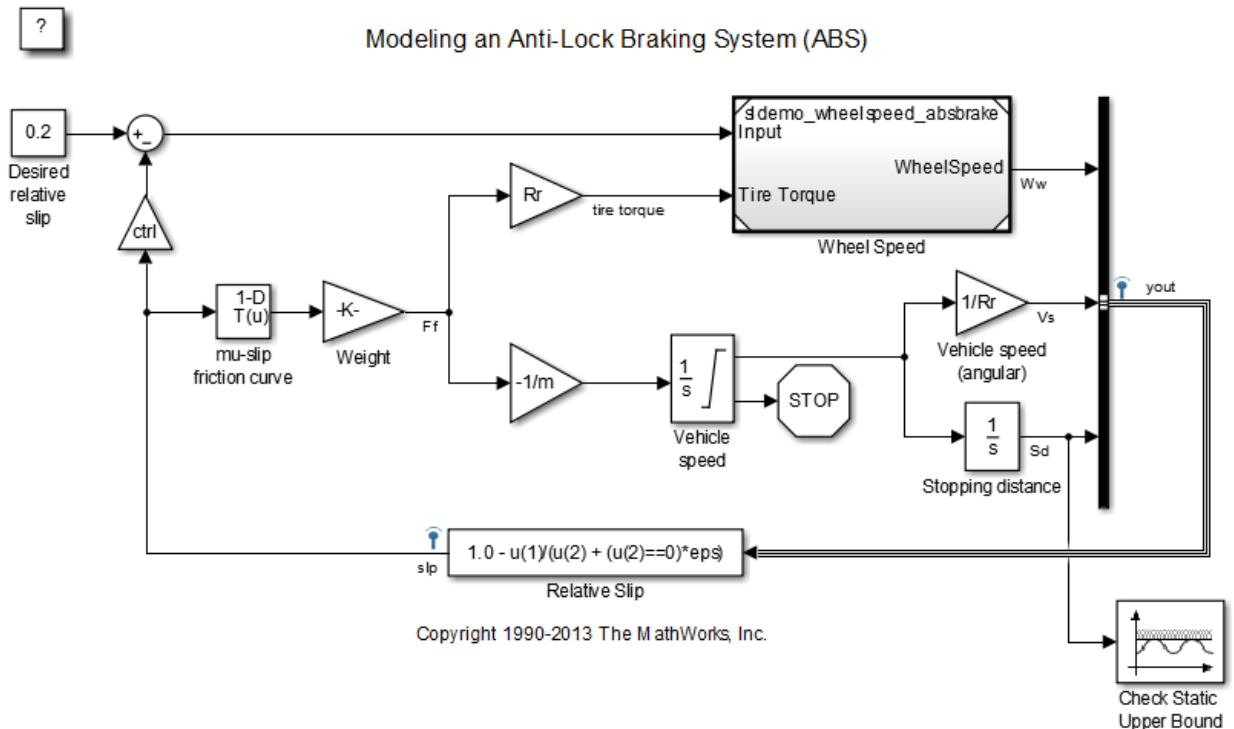
## Test a Simulation for Run-Time Errors

In this example, use a simulation test case with the `sldemo_absbrake` model to test for simulation run-time errors.

### Configure the Model

Configure the model to check if the stopping distance exceeds an upper bound.

- 1 Open the model `sldemo_absbrake`.
- 2 Add the Check Static Upper Bound block from the Model Verification library to the model.
- 3 Connect the Check Static Upper Bound block to the `Sd` signal.





- 4 In the Check Static Upper Bound block dialog box, and set **Upper bound** to 725.

## Create the Test Case

- 1 To open the Test Manager, from the model, select **Analysis > Test Manager**.
- 2 To create a test file, click **New**. Name and save the test file.

The new test file consists of a test suite that contains one baseline test case. They appear in the **Test Browser** pane.

- 3 Select **New > Simulation Test**.
- 4 Right-click the new simulation test case in the **Test Browser** pane, and select **Rename**. Rename the test case to Upper Bound Test.
- 5 In the test case, under **System Under Test**, click the **Use current model** button  to assign the sldemo\_absbrake model to the test case.
- 6 Under **Parameter Overrides**, click **Add** to add a parameter set.
- 7 In the dialog box, click the **Refresh** button  to update the model parameter list.
- 8 Select the check box next to the workspace variable m. Click **OK**.
- 9 Double-click the **Override Value** and enter 55.

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



## View Test Results

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

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

- 2 The result indicates a test failure. In this case, the stopping distance exceeded the upper bound of 725 and triggered an assertion from the Check Static Upper Bound block. The **Errors** section contains the assertion details.

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

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

## See Also

### More About

- “Run Tests in Multiple Releases” on page 6-52

## Automatically Create Test Cases for Your Model

<b>In this section...</b>
“Creating Test Cases from Model Elements” on page 6-16
“Generating Test Cases from a Model” on page 6-16

### Creating Test Cases from Model Elements

You can automatically create a set of test cases for your model by using the Test Manager. The Test Manager can automatically create test cases for these model elements:

- One (and only one) Signal Builder block at the top level of your model. One test case is created for each signal group in the Signal Builder block.
- Each test harness in your model. If a test harness contains one (and only one) Signal Builder block at the top level of test harnesses, a test case is created for each signal group in the Signal Builder block.

When you automatically create test cases, you specify whether the test case is a baseline, equivalence, or simulation test case. For more information on configuring baseline, equivalence, and simulation tests, see “Introduction to Test Manager” on page 5-2.

To automatically create test cases for your model:

- 1 In the Test Manager, select **New > Test File > Test File from Model**.
- 2 In the dialog box, select the model that you want to generate test cases from. The model must be on the MATLAB path.
- 3 Select the test case type, and click **Create**.

### Generating Test Cases from a Model

Generate test cases based on model hierarchy. Copyright 2015 The MathWorks, Inc.

This example shows how to generate test cases based on the components in your model. This example uses the model `sltestTestManagerCreateTestsExample`, which has been pre-configured with the following:

- Signal Builder group in the top model

- Test harnesses in the top model
- Signal Builder group at the top level of a test harness

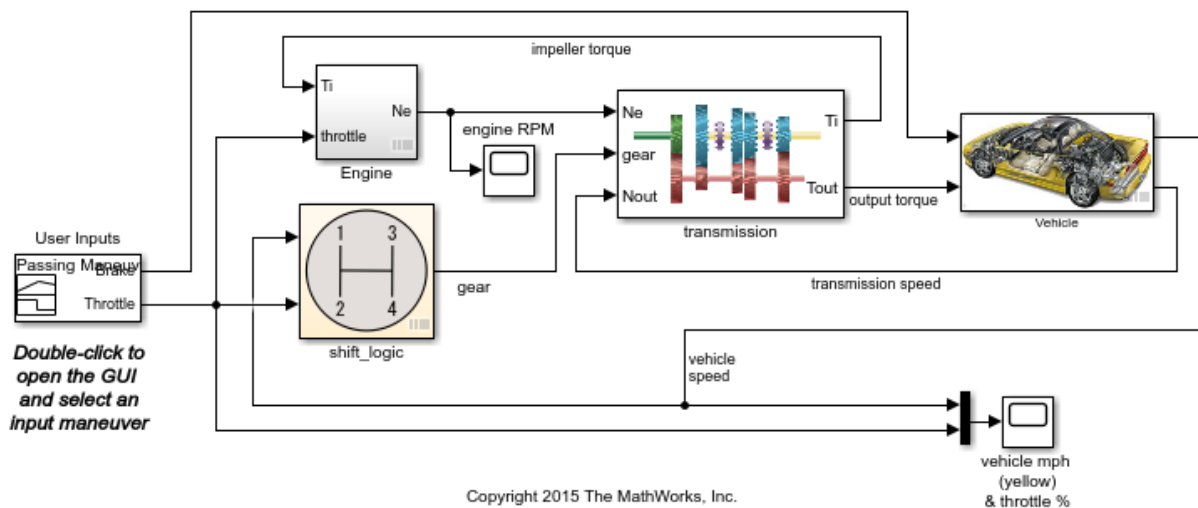
### Open the Model and Test Manager

Execute the following code to open the model configured with different components such as Signal Builder groups and test harnesses.

```
mdl = 'sltestTestManagerCreateTestsExample';
open_system(mdl);
```

### Generating Test Cases from a Model

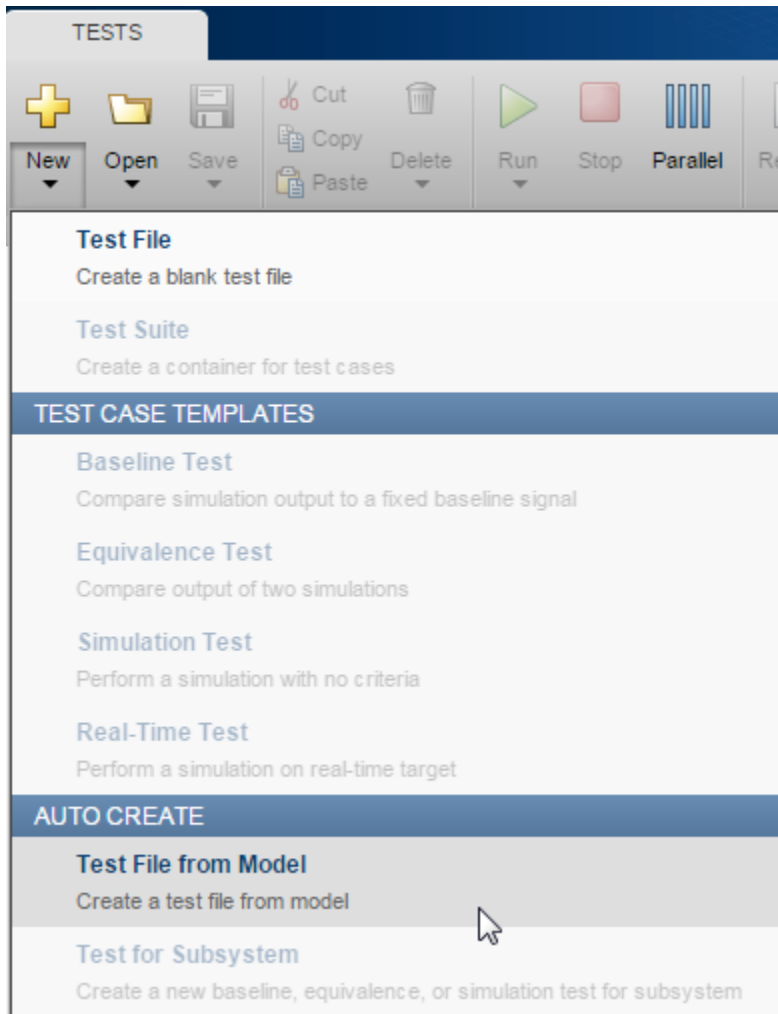
This example shows how to generate test cases based on the components in your model. To see the demo, execute `showdemo sltestTestManagerCreateTests` in MATLAB(R).



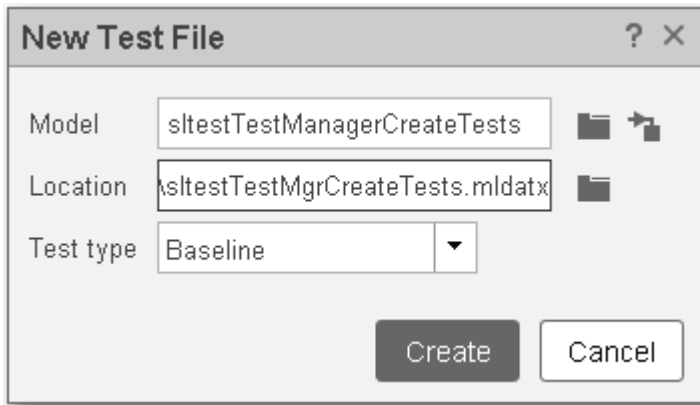
Open the test manager. Enter `sltestmgr` in the MATLAB command prompt.

### Generate Test Cases From the Model

In the test manager, click the **New** arrow and select **Test File > Test File from Model**.



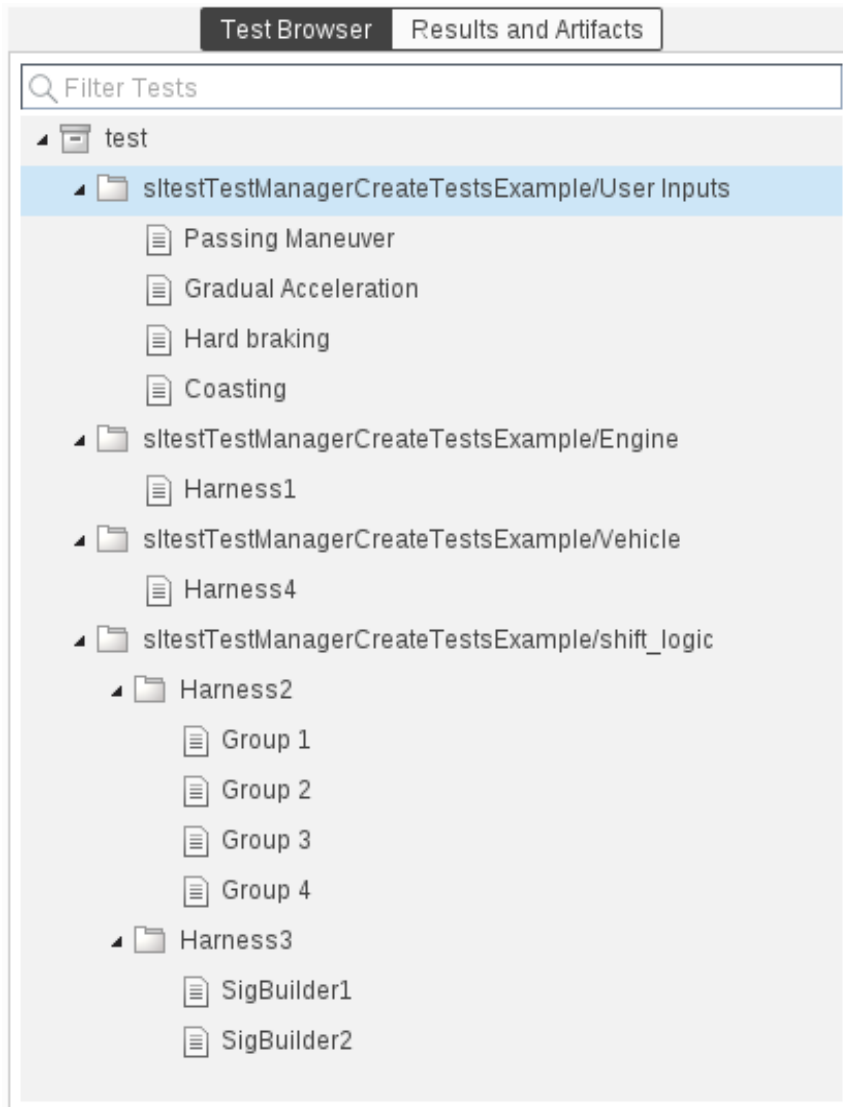
- 1 In the **New Test File** dialog box, click the **Use current model** button. to specify sltestTestManagerCreateTestsExample as the **Model**.
- 2 Specify the **Location** of the test file.
- 3 Select the **Baseline** from the **Test Type** dropdown. All test cases generated will be of the test type specified here.
- 4 Click **Create**.



The test manager creates a test case for each of the following:

- Signal Builder groups in the top model
- Test harnesses in the top model
- Signal Builder group at the top level of a test harnesses

In each generated test case, you need to specify the comparison criteria, equivalence or baseline, before you run the test.



```
close_system mdl, 0);  
clear mdl;
```

## See Also

### More About

- “Synchronize Test Cases to Model Changes” on page 6-24
- “Test Sections” on page 6-95
- “Test Model Output Against a Baseline” on page 6-9
- “Test a Simulation for Run-Time Errors” on page 6-13
- “Test Two Simulations for Equivalence”
- “Generate Tests for a Subsystem” on page 6-22

# Generate Tests for a Subsystem

Use the Test Manager to generate a test case for a subsystem. You can create a baseline, equivalence, or simulation test case. Generating the test case:

- Creates a test harness for the subsystem. The test harness provides a separate simulation environment from the main model.
- Simulates the main model and captures the subsystem input data from the main model simulation.
- For baseline tests, captures output data in a MAT-file or Microsoft® Excel® file. The output is the test case baseline data.
- Adds the input and output files to the test case.


After you create the test case, configure the test case if you need additional options such as coverage or reports.

You add a generated test case for a subsystem to a test file. If you need a test file to add the test case to, create one.

- Select the test file before you generate the test to add a test suite containing the test case.
- Select a test suite before you generate the test case to add the test case to the test suite. Or, select a test case in a test suite to add the test case to the test suite.

## Generate the Subsystem Test Case

You can save the input data to a MAT-file or Excel file. When creating a baseline test, selecting Excel saves the input and output data in the same file. For more information on using Excel files in Test Manager, see “Specify Microsoft Excel File Format for Signal Data” on page 6-25.

- 1 In the model that contains the subsystem that you want to create the test for, select the subsystem.
- 2 In Test Manager, select the test file or test suite that you want to create the test case in, and then select **New > Test for Subsystem**.
- 3 In the dialog box, click the **Use currently selected subsystem** button  to fill in the **Subsystem** and **Top model** fields.



- 4 Select the test type—baseline, equivalence, or simulation—and specify the file format.
- 5 Depending on the test type and the file format, you can specify the location for your inputs and your baseline outputs and the sheet name for Excel data.
- 6 Click **Create**. Test Manager adds a test harness to the subsystem and simulates the model.

After simulation completes, the test case includes inputs and, for baselines, outputs.

- For equivalence tests, inputs are added to the test case in the **Inputs** section under **Simulation 1**.
  - For simulation tests, inputs are added under **Inputs**.
  - For baseline tests, inputs are added under **Inputs** and outputs are added under **Baseline Criteria**.
- 7 Finish configuring the harness and test case for your test scenario.
  - 8 Save the model and the test case.

## See Also


### More About

- “Test Sections” on page 6-95
- “Test Model Output Against a Baseline” on page 6-9
- “Test a Simulation for Run-Time Errors” on page 6-13
- “Test Two Simulations for Equivalence”
- “Automatically Create Test Cases for Your Model” on page 6-16

# Synchronize Test Cases to Model Changes

If you change your model under test by adding Signal Builder groups or test harnesses, you can synchronize the test cases in a test file with your model. Also, if you remove model components, you can disable or delete test cases in the Test Manager when you synchronize.

Synchronizing your test cases generates one test for:

- Each new signal builder group in the Signal Builder block at the top level of your model and the top level of each harness. Your model must have only one Signal Block at those levels for Test Manager to generate test cases.
  - Each new harness in the model.
- 1 In the Test Manager **Test Browser** pane, hover over the test file name whose test cases you want to update.
  - 2 Click the synchronization button  next to the test file name.
  - 3 Follow the prompts to specify:
    - The type of test file to create for the new components
    - Whether to disable or delete out-of-date components

Disabled tests appear in the list in italic.

## See Also

### More About

- “Automatically Create Test Cases for Your Model” on page 6-16

## Specify Microsoft Excel File Format for Signal Data

You can specify signal data in a Microsoft Excel file to use as input to your test case or as baseline criteria (outputs). The Excel file includes time and signal data. To support a range of models and configurations, you can specify signal data of most data types. For exceptions, see “Limitations” on page 6-33. You can indicate whether signals are scalar, multidimensional, or complex. You can optionally specify the data type, block path and port index, units, interpolation type, and function-call execution times.

### Basic Excel File Format

The figure shows the basic format of the Excel file. This example uses scalar signals and the default data type, `double`, for all the signals.

	A	B	C	D
1	time	signal1	signal2	signal3
2	0	1	2	4
3	2	2	4	8
4	4	3	6	12
5	6	4	8	16

- When specifying time and signal data (and not function-call execution times), the `time` column is the first column. Time values must increase in value, and every cell must contain a value. Use the label `time`.
- Include one column for each input signal in the first row. In each column, include the signal data for each time point. Signal names are case-sensitive.
- If dataset elements have different time vectors, the spreadsheet can have more than one time column. In this case, the columns to the right of each time column, up to the next time column, define signals along that time vector. The signal columns must have the same number of rows as the time column they define values for. The figure shows an example that has a time column for each time vector.

	A	B	C	D	E	F
1	time	mySignal(1,3)	mySignal2	time	mySignal3	mySignal4
2	0	1	0.87	1.1	3	4
3	2	2	0.94	1.2	6	8
4	4	3	1.2	1.3	9	12
5	6	4	1.63			
6	8	5	1.8			

When you import data, you specify the mapping mode. To map using signal or block names, add the block or signal names and qualifiers in the first row. If you are mapping using block path and name, also specify them in the optional rows (see “Block Path and Port Index” on page 6-30). If you are mapping using port numbers, signal columns map to the model ports in order during import, ignoring the block or signal names.

## Input and Output Data

You can save inputs and outputs in the same Excel file in the same sheet. Specify whether the signals are for inputs or outputs in one of the optional rows, using **Source:Input** or **Source:Output** as the label. Keep all the inputs together and all the outputs together.

	A	B	C	D	E	F
1	time	Data1	Data2	time	out1	out2
2		Source:Input	Source:Input		Source:Output	Source:Output
3	1	2	0.23	1	2	0.23
4	2	3	1.23	2	3	1.23
5	3	4	2.23	3	4	2.23
6	4	5	3.23	4	5	3.23
7	5	6	4.23	5	6	4.23
8	6	7	5.23	6	7	5.23

To import the file as input data, use the **Inputs** section of the test case, described in “Use External Inputs in Test Cases” on page 6-42. To use the Excel file as expected outputs, select it to add as baseline data, in the **Baseline Criteria** section of the test case, described in “Baseline Criteria” on page 6-102.

When you capture inputs and expected outputs in Test Manager, you can save inputs and outputs to the same Excel file. Both sets of data are saved to the same sheet unless you specify a different sheet. Saving the inputs or expected outputs adds the file to the test. See “Capture Baseline Criteria” on page 6-102.

## Parameters

You can save parameter override values in the same Excel file in the same sheet. Specify the name of the parameter in the **Parameter:** column and the override value in the **Value:** column. Each row corresponds to a parameter. Enter vector and matrix parameters as you would in MATLAB. You can also use MATLAB expressions for parameter values. Values are read as strings and are evaluated at runtime. Signal data and parameter data always start on the same row.

time	Magnitude	Angle	Parameter:	Value:
	Source: Input			
0	1	90	Xscale	[1 2 3]
10	1	90	Yscale	1

The Parameter column must precede the Value column and both columns are required. For a parameter in a masked subsystem, add a third column MaskBlockPath and enter the path to the block:

Parameter:	Value:	MaskBlockPath:
Xscale	1	coordinate_transform/Scaling1
Yscale	1	coordinate_transform/Scaling1
Xscale	2	coordinate_transform/Scaling2
Yscale	2	coordinate_transform/Scaling2

## Signal Tolerances

You can specify tolerances for comparison signals in the Excel file. It is possible to define one or more tolerance types for each signal, in any order. Prefix the tolerance value with one of: "AbsTol:", "RelTol:", "LeadingTol:", "LaggingTol:", which correspond to absolute tolerance, relative tolerance, leading time tolerance, and lagging time tolerance. The format is <ToleranceType>:<ToleranceValue>.

time	point.pixels_x	point.pixels_y
	AbsTol: 0.001	AbsTol: 0.001
	RelTol: 0.001	RelTol: 0.001
	LeadingTol:0.0001	
		LaggingTol: 0.00001
	BlockPath: coordinate_transform/Bus Creator	BlockPath: coordinate_transform/Bus Creator
	Source: Output	
0	0	1

Tolerances are interpreted as floating-point doubles. Each tolerance type should be in a dedicated row. Once a row is declared to be a certain type of tolerance, all columns in

that row must be of that type. An empty cell is treated as tolerance of zero. For more information on tolerances, see “Test Model Output Against a Baseline” on page 6-9.

## Simulation for Equivalence Tests

When you perform equivalence tests in Test Manager, you compare the results of two simulations. If your Excel input file is for an equivalence test, you can specify the inputs for each simulation. Specify the simulation in one of the optional rows, using `Simulation:1` or `Simulation: 2` as the label. Keep the inputs for each simulation together.

	A	B	C	D	E	F
1	time	signal1	signal2	time	signal1	signal2
2	Simulation: 1			Simulation: 2		
3	0	2	1.7	0	4	2.3
4	1	3	4	1	5	6
5	2	4	7.3	2	6	9

## Scalar, Multidimensional, Complex, and Bus Signals

In addition to scalar signal names, you can indicate multidimensional, complex, and bus signals, or a combination of these. The figure shows some examples.

	A	B	C	D	E
1	time	signal1(1,3)	signal2 (real)	signal3(3) (imag)	mybus.x
2	0	1	2	1.1	4
3	2	2	4	1.2	8
4	4	3	6	1.3	12
5	6	4	8	1.4	16

Specify block paths, and optionally, port index in one of the optional rows. See “Block Path and Port Index” on page 6-30.

### Base Name

Names are case-sensitive.

### Multidimensional Signals

Use parentheses with the signal dimension after the signal name. For example:

- `mySignal(1,3)`

Dimensions on the signal that you do not specify default to zeros of the same data type and complexity as the dimensions that you specify.

### Complex Signals

For complex signals, use `(real)` or `(imag)` with the signal name. For example:

- `mySignal (real)`
- `mySignal (imag)`
- `mySignal(1,3) (imag)`

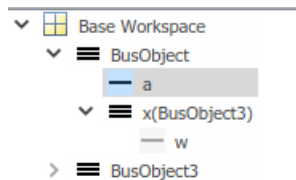
If you do not specify a real counterpart to an imaginary number, the real value defaults to zeros.

### Bus Signals

Specify bus signals in the form `signalname.busElement.nestedElement` for as many nested elements as the bus has. For example:

- `myBusSignal.x`
- `busSignal2.x.z`

Suppose the inport block `myBus` is a bus object with this structure:



In this case, specify the signal as `myBus.a.w`.

The figure shows an example of specifying bus signals with a bus data type (see “Data Type” on page 6-30). The `BusObj` data type also applies to the columns to the right because the base name for these signals is the same.

	A	B	C	D
1	time	myBus.x	myBus.x.y	myBus.x.y.z
2		Bus: BusObj		
3	0	1	4	1.1
4	0.2	3	8	1.2
5	0.4	5	12	1.3
6	0.6	7	16	1.4

### Block Path and Port Index

If you want to specify the signal using the block path, enter it in one of the optional rows in the form `BlockPath: path to block`. When you specify a block path, you can also specify the block port index. The default port index is 1. Enter the port index in the row following the block path in the form `PortIndex: port number`. For example:

- BlockPath: mymodel/myblock
- PortIndex: 2

	A	B
1	time	myblock
2		BlockPath: myModel/myblock
3		PortIndex: 2
4	0	1
5	2	2

### Data Type, Unit, Interpolation, and Block Path/Port Index

In the optional rows between the signal name and the time and signal data, you can include any combination of information about the signal:

- Data type
- Units
- Interpolation
- Block path and port index (see “Block Path and Port Index” on page 6-30)

#### Data Type

Enter data types in the row after the signal name. The default data type is `double`.



You can mix data types in the same row, but you must use the same data type for all columns of a multidimensional or complex signal.

You can leave the columns to the right of the data type declaration empty if that data type applies to the signal data in those columns. For example, here the data type `int16` applies to columns B and C because they are dimensions of the same signal.

	A	B	C	D
1	time	intsignal(1)	intsignal(2)	enumSignal
2		Type: int16		Enum: color
3	0	1	2	blue
4	0.2	2	4	green
5	0.4	3	6	yellow

### Built-In MATLAB Data Types

Specify built-in MATLAB data types supported in Simulink in the form `Type: data type`. See “Data Types Supported by Simulink” (Simulink). For example:

- Type: `int16`
- Type: `uint32`

### Enumerations

Specify an enumeration data type in the form `Enum: class`. For example:

- Enum: `school`

The data in the cells correspond to enumerated values. For example:

	A	B	C
1	time	signal1	signal2
2			Enum: color
3	0	1	blue
4	2	2	green
5	4	3	yellow
6	6	4	red

Enum data type dimensions that do not have data default to the default enumeration value.

### Fixed Point

Indicate a fixed-point data type using the prefix `Fixdt:`, followed by the data type in one of these forms:

- A `fixdt` constructor, for example, `Fixdt: fixdt(1,16)`.
- A unique data type name string, for example, `Fixdt: sfix16_B7`. To learn about specifying data type names, see “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).
- A `numericType` object in the base workspace, for example, `Fixdt: mytype`.

### Bus

Specify the bus object in the form `Bus: bus object` with a bus signal. For example:

- `Bus: BusObject1`

To specify a bus signal, see “Bus Signals” on page 6-29.

### Alias

Specify an alias data type in the form `Alias: alias type`. To learn about alias data types, see `Simulink.AliasType`.

### Units

Optionally, include a row for units. Specify units in the form `Unit: units`. You can specify units and physical quantity. For example:

- `Unit: g`
- `Unit: kg@mass`

### Interpolation

Optionally, include a row for interpolation. The default is linear. Specify interpolation as `Interp: zoh` or `Interp: linear`.

### Synchronization

Optionally, include a row for synchronization. The default is "union". Specify synchronization as `Sync: union` or `Sync: intersection`.

## Function-Call Execution Times

If the model contains control signals for function-call subsystems, add columns for each one before the first `time` column. Enter the control signal name in the column heading. Enter the points of time when you want to execute the function call in the column.

Function-call execution times that you specify are independent of the times in the `time` column. The figure shows how to format two function-call blocks that execute at various times. The time and signal data and data type information are independent of the function-call information.

	A	B	C	D	E
1	fcnCall1	fcnCall2	time	mysignal(1,3)	mysignal2
2				Type: int16	
3	0.1	0	0	1.1	6
4	0.1	2	1	1.6	7
5	0.8	4	2	2.1	8
6	1.3		3	2.6	9
7			4	3.1	10

## Limitations

Arrays of buses as a data type are not supported.

## See Also

`sltest.testmanager.BaselineCriteria` | `sltest.testmanager.TestInput`

## More About

- “Use External Inputs in Test Cases” on page 6-42
- “Baseline Criteria” on page 6-102

## Author a Test in Excel

### Generate an Excel Template

You can generate a template test spreadsheet from any model or harness (SUT). You can then complete the spreadsheet with external data and import into Simulink Test as a test case.

The Create Test from Spreadsheet wizard parses the SUT for test attributes and automatically generates a template spreadsheet and a test case:

- Inputs: Inputs are characterized by root input ports.
- Parameters: Named parameters in the model.
- Comparison signals: Logged signals and output ports.

The wizard allows you to filter and edit the attributes needed for testing. The resulting spreadsheet has separate column sets for inputs, parameters, and comparison signals. If multiple iterations are required, a separate sheet in the same file is generated for each scenario. You can expand the spreadsheet to add time-based signal data, tolerances, and parameter overrides. See “Specify Microsoft Excel File Format for Signal Data” on page 6-25 for the full description of the format readable by Simulink Test.

You can use the model `coordinate_transform_test` as an example for the process. The model must be on the MATLAB path.

Open the wizard. From Simulink Test Manager, select **New > Create Test from Spreadsheet**. Select **Create a test template file for specifying data** and follow the prompts.

In the **Attributes** page, select which attribute categories are to be included in the spreadsheet. For example, if no parameter overrides are necessary for the tests, clear Parameters. The attribute categories shown on the page are derived from the system under test. Comparison signals are always shown. If the test requires all attributes in a category as is, select **Yes, include all attributes in the spreadsheet** and click **Next**.

You can also filter attributes within attribute categories, and add signal tolerances for comparison.

- 1 On the **Attributes** page, select **No, I want to filter and edit the attributes**. This shows a page with a tab for each attribute category.

- 2 In the **Parameters** and **Comparison** tabs, clear the attributes that are not needed. For example, you can remove a logged signal from this list if it is not to be used for comparison in the tests.
- 3 You can change tolerances in the **Comparison** page. The tolerance settings apply to all signals in the list. To specify different tolerances for each signal, edit the spreadsheet after it is generated.

Inputs
Parameters
Comparison

Select signals below to participate in baseline comparisons

↻ Refresh

<input checked="" type="checkbox"/> NAME	INTERP
<input checked="" type="checkbox"/> pixels_x	linear
<input checked="" type="checkbox"/> pixels_y	linear

Tolerance settings (applies to all signals):

Absolute:  Relative:  Leading:  Lagging:

- 4 If you change the SUT during the selection process, click **Refresh** to synchronize the attribute lists with the SUT. Once selection is complete, click **Next** and keep following the prompts.

If any comparison signals are selected, the wizard runs the model to capture the baseline. Make sure that the model does not run indefinitely. The wizard creates two files:

- Excel spreadsheet — The spreadsheet includes columns for inputs, parameters, and comparison signals. Inputs and comparisons have different time bases. An identical sheet for each test scenario is generated. Complete the spreadsheet outside of MATLAB to uniquely define each scenario.

time	Magnitude	Angle	Parameter:	Value:	time	point.pixels_x	point.pixels_y
						AbsTol: 0.001	AbsTol: 0.001
						RelTol: 0.1	RelTol: 0.1
						BlockPath: coordinate_transform/Screen coordinates	BlockPath: coordinate_transform/Screen coordinates
	Source: Input					Source: Output	
0	0	0	Xscale	1	0	0	0
10	0	0	Yscale	1	0.2	0	0
					0.4	0	0
					0.6	0	0
					0.8	0	0

- Test file — The test case imports the Excel spreadsheet. The fields defined in the spreadsheet are locked to the spreadsheet, and cannot be edited in the Test Manager:

## New Test Case 1

 Enabled

[coordinate\\_test\\_signals](#) » [New Test Suite 1](#) » [New Test Case 1](#)

Baseline Test

Select releases for simulation:

Create Test Case from External File

File:



▶ TAGS

▼ PARAMETER OVERRIDES\*

?

PARAMETER SET / WORKSPACE VARIABLE	SHEETS	OVERRIDE VALUE	SOURCE	MODEL ELEMENT	+
▼ <input type="checkbox"/> coordinate_test_signals.xlsx	Scenario1				▲
<input checked="" type="checkbox"/> Xscale		1			
<input checked="" type="checkbox"/> Yscale		1			
▼ <input type="checkbox"/> coordinate_test_signals.xls...	Scenario2				
<input checked="" type="checkbox"/> Xscale		1			
<input checked="" type="checkbox"/> Yscale		1			

To change the locked fields, edit the spreadsheet outside of MATLAB. If you change a parameter, you must capture the baseline again by clicking Capture...

▼ BASELINE CRITERIA\*

Include baseline data in test result

SIGNAL NAME	SHEETS	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL	+
▶ <input type="checkbox"/> Scenario1	Scenario1	0	0.00%	0	0	
▶ <input type="checkbox"/> Scenario2	Scenario2	0	0.00%	0	0	
▶ <input type="checkbox"/> Scenario3	Scenario3	0	0.00%	0	0	

+ Add... ↩ Capture... ✎ Edit... ↻ Refresh Visualize Delete

## See Also

### Related Examples

- “Specify Microsoft Excel File Format for Signal Data” on page 6-25
- “Create Test with Data from Excel” on page 6-39



## Create Test with Data from Excel

### In this section...

“Create a Test Case from an Excel Spreadsheet” on page 6-39

“Import an Excel Spreadsheet into a Test Case” on page 6-40

You can use test data in Excel to create a test case or populate an existing test case. The spreadsheet must be in the format given in “Specify Microsoft Excel File Format for Signal Data” on page 6-25.

### Create a Test Case from an Excel Spreadsheet

You can create a test case in Test Manager using the Create Test from Spreadsheet wizard. From Simulink Test Manager, select **New > Create Test from Spreadsheet**. Select **Use existing test data from a spreadsheet** and follow the prompts. you can use the following spreadsheet and model as an example:

```
<matlabroot>\examples\simulinktest\coordinate_tests.xlsx  
coordinate_transform_test.slx
```

In the **Attributes** page, make sure all attribute categories that exist in the spreadsheet are displayed. Click **Validate** to map each input to the model by block name. If necessary, make changes to the spreadsheet and/or SUT and click **Refresh** and validate again. After a successful validation, save the test.

- The test case imports the spreadsheet. The fields defined in the spreadsheet are locked to the spreadsheet, and cannot be edited in the Test Manager:

## New Test Case 1

 Enabled[coordinate\\_test\\_signals](#) » [New Test Suite 1](#) » [New Test Case 1](#)

Baseline Test

Select releases for simulation:  Create Test Case from External FileFile: 

TAGS

## PARAMETER OVERRIDES\*

?

PARAMETER SET / WORKSPACE VARIABLE	SHEETS	OVERRIDE VALUE	SOURCE	MODEL ELEMENT	
<input type="checkbox"/> coordinate_test_signals.xlsx	Scenario1				+
<input checked="" type="checkbox"/> Xscale		1			▲
<input checked="" type="checkbox"/> Yscale		1			
<input type="checkbox"/> coordinate_test_signals.xls...	Scenario2				
<input checked="" type="checkbox"/> Xscale		1			
<input checked="" type="checkbox"/> Yscale		1			

To change the locked fields, edit the spreadsheet outside of MATLAB.

## Import an Excel Spreadsheet into a Test Case

If you have a test case and want to add test data to it from Excel spreadsheet, you must associate it with the spreadsheet:

- 1 Open the test case.
- 2 Check the **Create Test Case from External File** option.
- 3 Browse for the spreadsheet with the test data.

The input, parameter, and comparison signal data in the spreadsheet overrides the data in the test case. The fields defined in the spreadsheet are locked to the spreadsheet. To edit, do one of the following:

- Edit the spreadsheet outside of MATLAB and click **Refresh** for the **File** field.
- Clear the **Create Test Case from External File** option and edit the test case in the Test Manager. Note that selecting this option again causes values in the spreadsheet to overwrite the values in the test.

## See Also

“Specify Microsoft Excel File Format for Signal Data” on page 6-25 | “Author a Test in Excel” on page 6-34

## Use External Inputs in Test Cases

### In this section...

“Mapping Status” on page 6-42

“Add a MAT-File as an External Input” on page 6-43

“Add Microsoft Excel File as Input” on page 6-43

“Importing Microsoft® Excel® Data” on page 6-44

You can use external model inputs from MAT-files or Microsoft Excel files in a test case. When you add the file, you can select one of these mapping modes to specify the model element to map to:

- The names of the inport block the signal data corresponds to
- The full block path name, that is, in the form `system/block`
- The name of the signal associated with the inport block
- Port number, that is, sequential port numbers of the inport blocks, starting at 1

You can add multiple external input files to a test case. After you add the files, select the one you want to use in the test case from the **External Inputs** table. If you are using test iterations, you can assign one input file to each iteration.

For more information about how Simulink handles inport mapping, see “Map Root Inport Signal Data” (Simulink).

### Mapping Status

When you map external inputs to model elements, the mapping can create these possible results. These results appear under **Inputs** in the Test Manager interface in the **Status** column:

- Mapped — The mapping succeeded and no further action is needed.
- Failed — The mapping failed. Click the **Failed** link for more information.
- Warning — The mapping occurred with warnings. Click the **Warning** link to see whether you need to address them
- Stale — This status can occur when you update your external inputs in Test Manager. A stale state occurs if you did not map the new inputs. To address this status, click the

**Status** link, which opens the Add Input dialog box. Click **Map Inputs** to map the new input data and then click **Add**.

## Add a MAT-File as an External Input

- 1 In the test case, expand the **Inputs** section and click **Add**.
- 2 Browse to the MAT-file and click **Add**.
- 3 Under **Input Mapping**, choose a mapping mode.
- 4 Click **Map Inputs**. The **Mapping Status** table shows the port and signal mapping.

For information about troubleshooting the mapping status, see “Understand Mapping Results” (Simulink).

- 5 Click **Add**.

## Add Microsoft Excel File as Input

You can import Microsoft Excel spreadsheets to use as inputs. You can import multiple sheets at once and specify a range of data. Selecting sheets and specifying ranges is useful when each sheet contains a different data set or the same file contains input data and expected outputs.

For information about the Excel file format, see “Specify Microsoft Excel File Format for Signal Data” on page 6-25.

- 1 In the test case, expand the **Inputs** section and click **Add**.
- 2 Browse to your Microsoft Excel file and click **Add**.
- 3 Select each sheet that contains input data. You can specify a range of data.
- 4 If you want to use each sheet to create an input set in the table, select **Create scenarios from each sheet**.
- 5 Under **Input Mapping**, select a mapping mode.
- 6 Click **Map Inputs**. The **Mapping Status** table shows the port and signal mapping.

For more information about troubleshooting the mapping, see “Understand Mapping Results” (Simulink).

- 7 Click **Add**.

## Importing Microsoft® Excel® Data

Test a model using inputs stored in Microsoft Excel.

This example shows how to create a test case in the Test Manager and map data to the test case from a Microsoft® Excel® file. Input mapping supports Microsoft Excel spreadsheets only for Microsoft Windows®.

### Create a Test File

1. Open the Test Manager. Enter

```
sltest.testmanager.view
```

2. In the test manager toolbar, select **New > Test File**. Save the file to a writable directory. The test manager creates a test file with an empty baseline test case.

3. In the test browser, select the test case. In the test editor, under the **System Under Test** section, enter `sltestExcelExample`.



### Configure the External Inputs.

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

2. To include the input data in the test results, click **Include external inputs/signal builder data in test result**.

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

4. In the **Add Input** dialog box, for **File**, select the `sltestExampleInputs.xlsx` from the `matlab/toolbox/simulinktest/simulinktestdemos` directory.

5. In the **Add Input** dialog box,

- Select the **Acceleration** sheet from the sheets table.
- Select **Mapping Mode** : Block Name.

- Click **Map Inputs**.
- Click **Add**.

**Add Input** ? X

**INPUT FILE SPECIFICATION**

File:

Add iterations to run this input

▶ SHEETS AND RANGE SPECIFICATION

▼ INPUT MAPPING

Mapping Mode:  Map Inputs

Compile the system under test

▼ MAPPING STATUS

*Successfully mapped inputs.*

PORT	BLOCK NAME	MAPPED SIGNAL	STATUS
1	sltestExcelExample/Throttle	Throttle	✔
2	sltestExcelExample/Brake	BrakeTorque	✔

▶ ADVANCED

Add
Cancel

The **Mapping Mode** controls the method used to map data from the Microsoft Excel sheet to root-level Inport blocks in the model. For more information, see Use External Inputs in Test Cases.

The test case shows the inputs mapped.

▼ INPUTS\*

Include external inputs/signal builder data in test result

Stop simulation at last time point

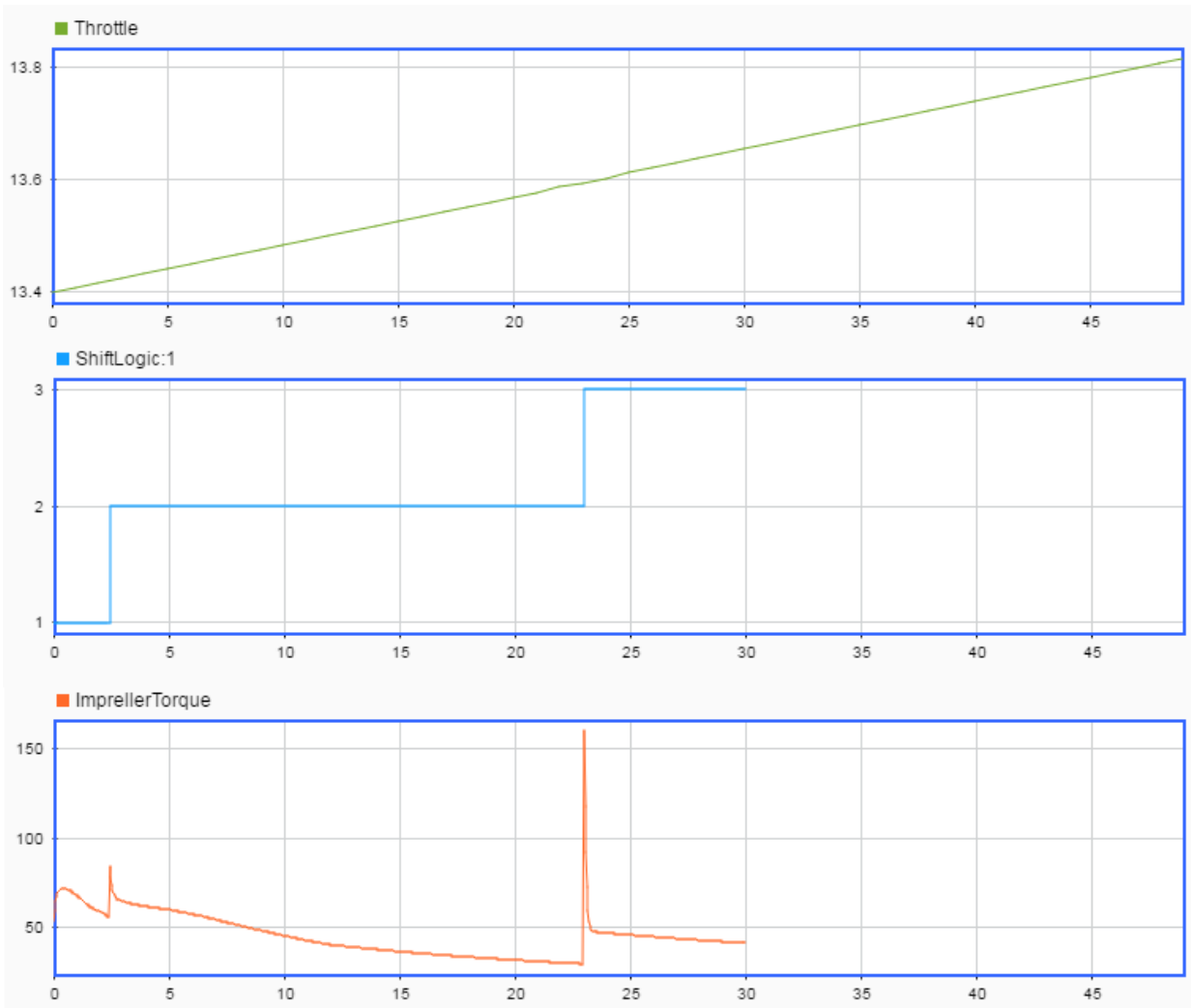
EXTERNAL INPUTS

NAME	FILE	SHEET	STATUS
<input checked="" type="checkbox"/> sltestExampleInputs.xlsx	T:\	<a href="#">Acceleration</a>	<a href="#">Mapped</a>

### Run the Test

1. In the toolbar, click **Run**.
2. In the **Results and Artifacts** pane, you can plot signals from the external inputs or the simulation output.





## See Also

`sltest.testmanager.TestInput`

### **More About**

- “Map Signal Data to Root Inports” (Simulink)
- “Map Root Inport Signal Data” (Simulink)
- “Specify Microsoft Excel File Format for Signal Data” on page 6-25

## Create Data Files to Use as Test Inputs

You can use Test Manager to create MAT-file and Microsoft Excel data files to use as inputs to test cases. You generate a template that contains the signal names and the times and then enter the data.

Creating a data file also adds the file to the list of available input files for the test case. After you add input data, you can then select the file to use in your test case.

You can create files for input data only for tests that run in the current release. To select the release, in the test case, use the **Select releases for simulation** list.

You can edit input files. After you create the template, select the file from the list of input files and click **Edit**. MAT-files open in the signal editor. Excel files open in Excel.

Selecting the **Add an iteration that runs this input** check box adds an iteration to the test case under **Table Iterations** and assigns the input file to it. After you create the input file, continue to specify the iteration. For more information on iterations, see “Run Combinations of Tests Using Iterations” on page 6-70.

### Create a Microsoft Excel File for Input Data

Creating a Microsoft Excel file for input data adds the signal names from the model and data type information for each signal to the spreadsheet, along with the times, based on the input blocks in the model. Selecting the **Add an iteration that runs this input** button adds a sheet for the iteration.

To learn about the Excel file format, see “Specify Microsoft Excel File Format for Signal Data” on page 6-25.

- 1 In the test case, under **System Under Test**, specify the model whose input data you want to create an Excel file for.
- 2 In the **Inputs** section of the test case, click **Create**.
- 3 In the dialog box, set the file format to Excel. Specify the location for the file.
- 4 Click **Create**.

The Excel file opens. It includes time and signal columns based on the input signals for the test case. If you used the same file to capture your baselines to, the outputs and inputs both appear in the Excel file in the same sheet.

- 5 In each column, add the time and signal values.

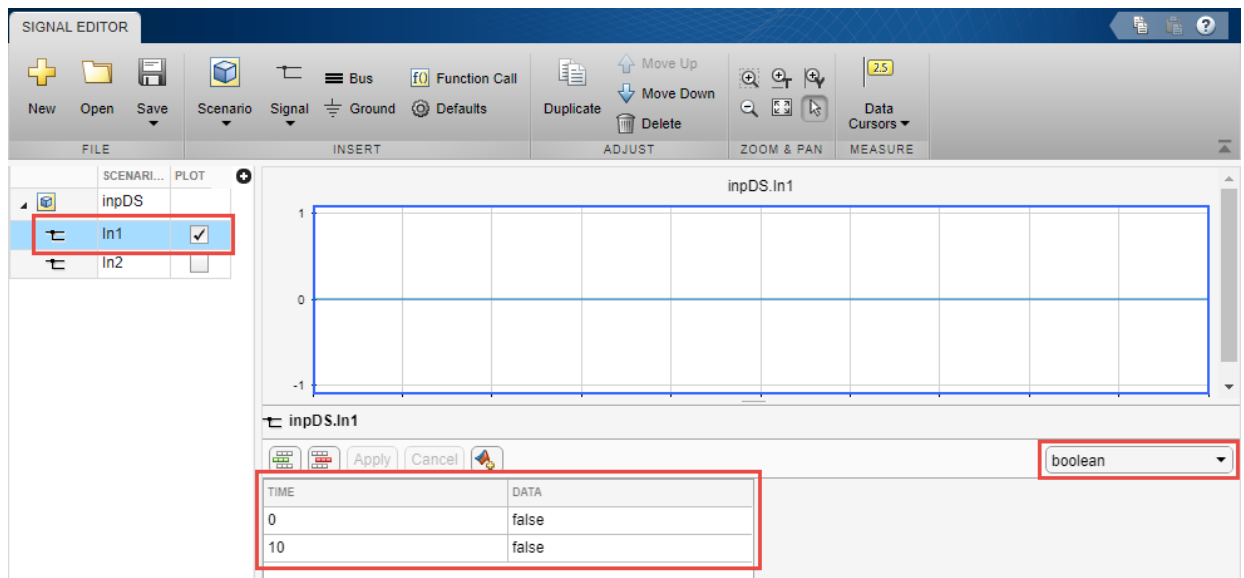
- 6 Save the Excel file.

## Create a MAT-File for Input Data

- 1 In the test case, under **System Under Test**, specify the model whose input data you want to create a MAT-file for.
- 2 In the **Inputs** section of the test case, click **Create**.
- 3 In the dialog box, set the file format to MAT - file. Specify the location for the MAT-file and click **Create**.

The signal editor opens.

- 4 In the **Scenarios and Signals** pane of the signal editor, expand the data node. Then select the signal whose data you want to add.
- 5 Specify the signal data. Select the data type from the list, and enter the time and signal data for the signal.



- 6 To update your signal data, click **Apply**.
- 7 After adding the signal data, click **Save**.

## See Also

### More About

- “Select Releases for Testing” on page 6-95
- “Specify Microsoft Excel File Format for Signal Data” on page 6-25
- “Use External Inputs in Test Cases” on page 6-42
- “Inputs” on page 6-100

## Run Tests in Multiple Releases

If you have more than one release of MATLAB installed, you can run tests in multiple releases. This option lets you run tests in releases that do not have Simulink Test, starting with R2011b.

While you can run test cases on models in previous releases, the release you run the test in must support the features of the test. If, for example, your test involves test harnesses or test sequences, the release must support those features for the test to run.

Before you can create tests that use additional releases, add them to your list of available releases using Test Manager preferences. See “Add Releases Using Test Manager Preferences” on page 6-53.

### Considerations for Testing in Multiple Releases

#### Testing Models in Previous or Later Releases

Your model or test harness must be compatible with the MATLAB version running your test.

- If you have a model created in a newer version of MATLAB, to test the model in a previous version of MATLAB, export the model to a previous version and simulate the exported model with the previous MATLAB version. For more information, see the information on exporting a model in “Save a Model” (Simulink).
- To test a model in a more recent version of MATLAB, consider using the Upgrade Advisor to upgrade your model for the more recent release. For more information, see “Consult the Upgrade Advisor” (Simulink).

#### Test Case Compatibility with Previous Releases

When performing testing in multiple releases, the MATLAB version must support the features of your test case. Previous MATLAB versions do not support test case features unavailable in that release. For example:

- Test harnesses are supported for R2015a and later.
- The Test Sequence block is supported for R2015a and later.
- `verify()` statements are supported for R2016b and later.

## Test Case Limitations with Multiple Release Testing

Certain features are not supported for multiple release testing:

- Parallel test execution
- Running test cases with the MATLAB Unit Test framework
- Real-time tests
- Input data defined in an external Excel document
- Coverage collection in the Test Manager
- Generating additional tests using Simulink Design Verifier to increase coverage
- Including custom figures from test case callbacks

## Add Releases Using Test Manager Preferences

Use a Test Manager preference to add to the list of release to run tests in. You can delete a release that you added to the list. You cannot delete the release from which you are running Test Manager.

- 1 In the Test Manager toolstrip, click **Preferences**.
- 2 In the **Preferences** dialog box, click **Release**. The **Release** pane lists the release you are running Test Manager from.
- 3 In the **Release** pane, click **Add**.
- 4 Browse to the location of the MATLAB release you want to add and click **OK**.

## Run Baseline Tests in Multiple Releases

When you run a baseline test with Test Manager set up for multiple releases, you can:

- Create the baseline in the release you want to see the results in, for example, to try different parameters and apply tolerances.
- Create the baseline in one release and run it in another release. Using this approach you can, for example, know whether a newer release produces the same simulation outputs as an earlier release.

Create the baseline.

- 1 Make sure that the release has been added to your Test Manager preferences.

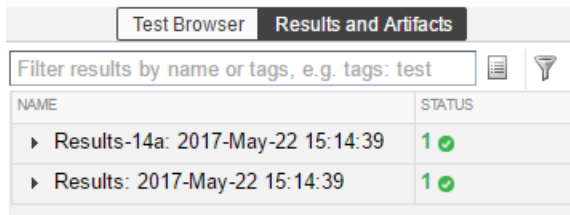
- 2 Create a test file, if necessary, and add a baseline test case to it.
- 3 In the test case, from the **Select release for simulation** list, select the releases you want to run the test case in.
- 4 Under **System Under Test**, enter the name of the model you want to test.
- 5 Set up the rest of the test.
- 6 Capture the baseline. Under **Baseline Criteria**, click **Capture**.
- 7 Select the release you want to use for the baseline simulation. Specify the file format and save and name the baseline.

For more information about capturing baselines, see “Capture Baseline Criteria” on page 6-102.



After you create the baseline, you can run the test in a release available in the Test Manager. Each release you select generates a set of results.

- 1 In the test case, set **Select releases for simulation** to the releases you want to use to compare against your baseline. For example, select only the release for which you created the baseline to perform a baseline comparison against the same release.
- 2 Specify the test options.
- 3 From the toolstrip, click **Run**.

For each release that you select when you run the test case, pass-fail results appear in the **Results and Artifacts** pane. For results from a release other than the one you are running Test Manager from, the release number appears in the name.



The screenshot shows the 'Results and Artifacts' pane in the Test Manager. At the top, there are two tabs: 'Test Browser' and 'Results and Artifacts'. Below the tabs is a search bar with the text 'Filter results by name or tags, e.g. tags: test' and a filter icon. Below the search bar is a table with two columns: 'NAME' and 'STATUS'. The table contains two rows of results, both with a status of '1' and a green checkmark icon.

NAME	STATUS
▶ Results-14a: 2017-May-22 15:14:39	1 
▶ Results: 2017-May-22 15:14:39	1 

## Run Equivalence Tests in Multiple Releases

When you run an equivalence test, you compare two simulations from the same release to see if differences in the simulations are within the specified tolerance.



- 1 Make sure that the release has been added to your Test Manager preferences.
- 2 Create a test file, if necessary, and add an equivalence test case to it.
- 3 In the test case, from the **Select release for simulation** list, select the releases you want to run the test case in.
- 4 Under **System Under Test**, enter the model you want to test.
- 5 Set the values under **Simulation 1** and **Simulation 2** to use as the basis for testing.
- 6 To set tolerances for the logged signals, under **Equivalence Criteria**, click **Capture**. Select the release you want to use for capturing the signals, and click **OK**. Clicking **Capture** copies the list of the signals being logged in Simulation 1. Then set the tolerances as desired.
- 7 In the toolbar, click **Run**.

The test runs for each release you selected, running the two simulations in the same release and comparing the results for equivalence. For each release that you selected when you ran the test case, pass-fail results appear in the **Results and Artifacts** pane. For results from a release other than the one you are running Test Manager from, the release number appears in the name.

Test Browser		Results and Artifacts	
Filter results by name or tags, e.g. tags: test			
NAME		STATUS	
▶ Results-14a: 2017-May-22 15:14:39		1	🟢
▶ Results: 2017-May-22 15:14:39		1	🟢

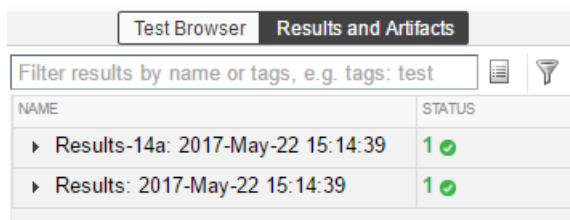
## Run Simulation Tests in Multiple Releases



Running a simulation test simulates the model in each release you select using the criteria you specify in the test case.

- 1 Make sure that the release has been added to your Test Manager preferences.
- 2 Create a test file, if necessary, and add a simulation test case template to it.
- 3 In the test case, from the **Select release for simulation** list, select the releases you want to run the test case in.
- 4 Under **System Under Test**, enter the model you want to test.

- 5 Under **Simulation Outputs**, select the signals to log.
- 6 In the toolbar, click **Run**.

The test runs, simulating for each release you selected. For each release, pass-fail results appear in the **Results and Artifacts** pane. For results from a release other than the one you are running Test Manager from, the release number appears in the name.



NAME	STATUS
▶ Results-14a: 2017-May-22 15:14:39	1 
▶ Results: 2017-May-22 15:14:39	1 

## See Also

`sltest.testmanager.getpref` | `sltest.testmanager.setpref`

## More About

- “Select Releases for Testing” on page 6-95

## Examine Test Failures and Modify Baselines

After you run a baseline test in the Test Manager, you can update the baseline. For example:

- If you changed your model, you can use the new simulation output as the baseline. You can examine the failures that occurred because of the differences and update the baseline with part or all of the new output. See “Examine Test Failure Signals and Update Baseline Test” on page 6-57.
- If your test plan changed and you expect different outputs, you can manually edit the time points. See “Manually Update Signal Data in a Baseline” on page 6-60.

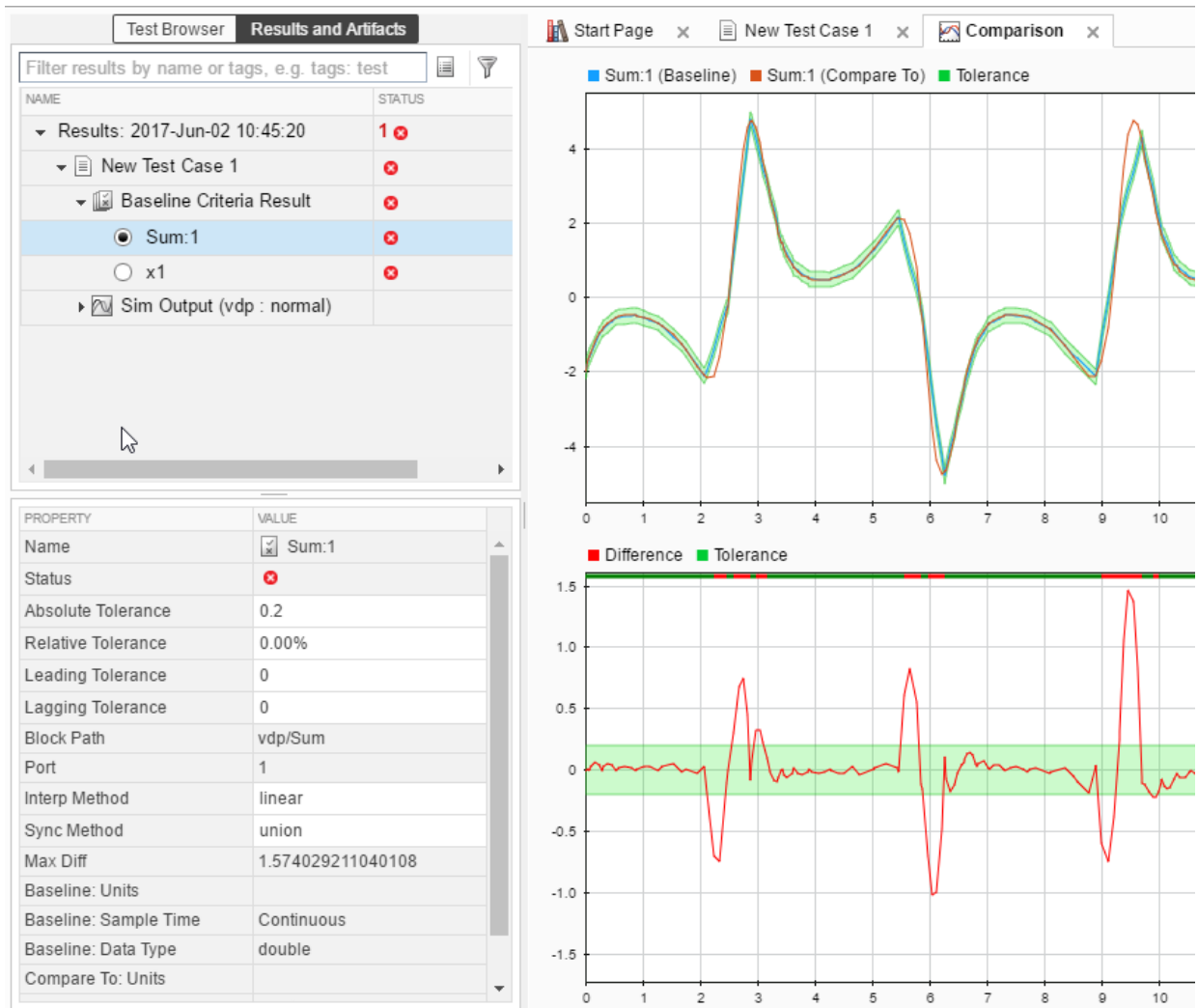
### Examine Test Failure Signals and Update Baseline Test

Suppose that you run a test against a baseline and the result does not match the baseline, causing test failure. It is possible that the newer simulation better represents your desired test results or that some of the points of failure are your preferred results. You can examine the signal and failures in the data inspector view in Test Manager and decide whether you want to update the baseline or sections of the baseline.

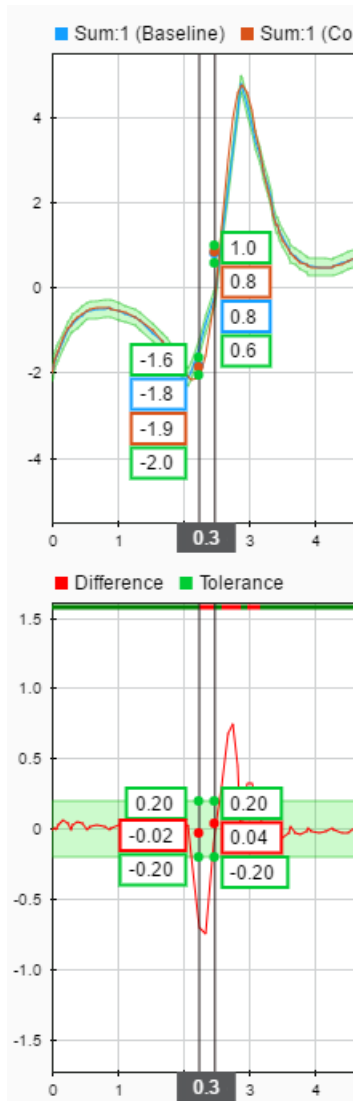
Suppose that your model uses a new solver. When you run the test case, the results do not match, causing the test to fail.

- 1 Open the test file that contains the baseline test case you want to run.
- 2 Select the test case and run it.
- 3 If the test fails, in the **Results and Artifacts** pane, expand the Baseline Criteria. Select a signal that failed that you want to examine.

When you select the signal, the data inspector view opens. The top graph is the baseline simulation signal overly. The bottom is the difference between those signals and the tolerance. You can adjust tolerances in the pane in the lower-left corner of the Test Manager. This example shows an absolute tolerance of .2.



- To examine each failure, in the toolstrip, click **Next Failure** or **Previous Failure**. Each contiguous set of failed signal comparison points makes up one region. Data cursors show the bounds of each region.



- 5 You can update the baseline data to use newer simulation results using the **Update Baseline** menu.
- To update the entire signal, select **Update Baseline Signal**.

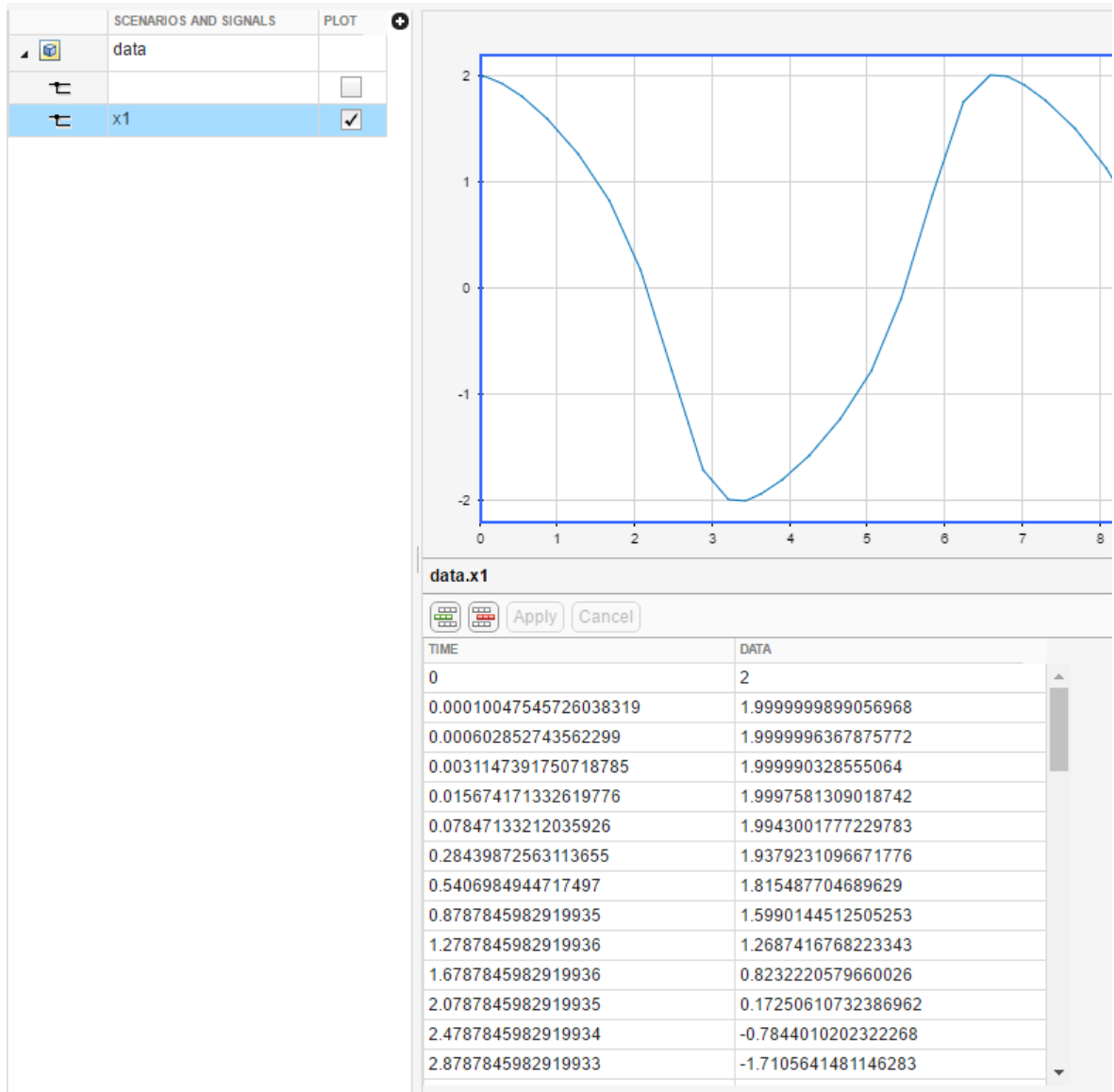
- To update only the data in the failure region, select **Update Selected Signal Region**.
- To replace all the signal data in the baseline with the new data, select **Update All Signals**.

### Manually Update Signal Data in a Baseline

If your model changes such that you expect a different simulation output, you can update all or part of the baseline signal data. If the baseline is a MAT-file, you can edit the data in the signal editor. Microsoft Excel files open in Excel.

To update signal data in a MAT-file baseline:

- 1 Open the test file that contains the baseline you want to edit.
- 2 Select the test case.
- 3 Under **Baseline Criteria**, select the baseline whose signal data you want to edit. Click **Edit**.
- 4 The signal editor opens. In the **Scenarios and Signals** pane, expand the **data** node.
- 5 Select the check box next to the signal whose data you want to edit.



---

**Tip** To see the time and data for points, display a data cursor and drag it along the signal.

---

- 6 Edit the signal data in the table, and then click **Apply**.
- 7 To update the baseline with the new expected output data, click **Save**.

## See Also

### More About

- “Work with Basic Signal Data” (Simulink)
- “Inspect Simulation Data” (Simulink)
- “Test Model Output Against a Baseline” on page 6-9



## Automate Tests Programmatically

In this section...
"List of Functions and Classes" on page 6-63
"Create and Run a Baseline Test Case" on page 6-65
"Create and Run an Equivalence Test Case" on page 6-66
"Run a Test Case and Collect Coverage" on page 6-67
"Create and Run Test Case Iterations" on page 6-68

### List of Functions and Classes

Function	Description
<code>sltest.testmanager.view</code>	Open the Simulink Test Manager
<code>sltest.testmanager.createTestsFromModel</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 test 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 test files from the Simulink Test Manager
<code>sltest.testmanager.close</code>	Close the Simulink Test Manager
<code>sltest.testmanager.clearResults</code>	Clear results from the Simulink Test Manager
<code>sltest.testmanager.importResults</code>	Import Test Manager results file

<b>Function</b>	<b>Description</b>
<code>sltest.testmanager.exportResults</code>	Export results set from Test Manager
<code>sltest.testmanager.getResultSets</code>	Returns result set objects in Test Manager

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

## Create and Run a Baseline Test Case

This example shows how to use `sltest.testmanager` functions, classes, and methods to automate tests and generate reports. You can create a test case, edit the test case criteria, run the test case, and generate results reports programmatically. The example compares the simulation output of the model to a baseline.

```
% Create the test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('API Test File');
ts = createTestSuite(tf,'API Test Suite');
tc = createTestCase(ts,'baseline','Baseline API Test Case');

% Remove the default test suite
tsDel = getTestSuiteByName(tf,'New Test Suite 1');
remove(tsDel);

% Assign the system under test to the test case
setProperty(tc,'Model','sldemo_absbrake');

% Capture the baseline criteria
baseline = captureBaselineCriteria(tc,'baseline_API.mat',true);

% Test a new model parameter by overriding it in the test case
% parameter set
ps = addParameterSet(tc,'Name','API Parameter Set');
po = addParameterOverride(ps,'m',55);

% Set the baseline criteria tolerance for one signal
sc = getSignalCriteria(baseline);
sc(1).AbsTol = 9;

% Run the test case and return an object with results data
ResultsObj = run(tc);

% Open the Test Manager so you can view the simulation
% output and comparison data
sltest.testmanager.view;

% Generate a report from the results data
filePath = 'test_report.pdf';
sltest.testmanager.report(ResultsObj,filePath,...
    'Author','Test Engineer',...
    'IncludeSimulationSignalPlots',true,...
    'IncludeComparisonSignalPlots',true);
```

The test case fails because only one of the signal comparisons between the simulation output and the baseline criteria is within tolerance. The results report is a PDF and opens when it is completed. For more report generation settings, see the `sltest.testmanager.report` function reference page.

## Create and Run an Equivalence Test Case

This example compares signal data between two simulations to test for equivalence.

```
% Create the test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('API Test File');
ts = createTestSuite(tf, 'API Test Suite');
tc = createTestCase(ts, 'equivalence', 'Equivalence Test Case');

% Remove the default test suite
tsDel = getTestSuiteByName(tf, 'New Test Suite 1');
remove(tsDel);

% Assign the system under test to the test case
% for Simulation 1 and Simulation 2
setProperty(tc, 'Model', 'sldemo_absbrake', 'SimulationIndex', 1);
setProperty(tc, 'Model', 'sldemo_absbrake', 'SimulationIndex', 2);

% Add a parameter override to Simulation 1 and 2
ps1 = addParameterSet(tc, 'Name', 'Parameter Set 1', 'SimulationIndex', 1);
po1 = addParameterOverride(ps1, 'Rr', 1.20);

ps2 = addParameterSet(tc, 'Name', 'Parameter Set 2', 'SimulationIndex', 2);
po2 = addParameterOverride(ps2, 'Rr', 1.24);

% Capture equivalence criteria
eq = captureEquivalenceCriteria(tc);

% Set the equivalence criteria tolerance for one signal
sc = getSignalCriteria(eq);
sc(1).AbsTol = 2.2;

% Run the test case and return an object with results data
ResultsObj = run(tc);

% Open the Test Manager so you can view the simulation
% output and comparison data
sltest.testmanager.view;
```

In the Equivalence Criteria Result section of the Test Manager results, the `yout.Ww` signal passes because of the tolerance value. The other signal comparisons do not pass, and the overall test case fails.

## Run a Test Case and Collect Coverage

This example shows how to use a simulation test case to collect coverage results. To collect coverage, you need a Simulink Coverage license.

```
% Create the test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('API Test File');
ts = createTestSuite(tf, 'API Test Suite');
tc = createTestCase(ts, 'simulation', 'Coverage Test Case');

% Remove the default test suite
tsDel = getTestSuiteByName(tf, 'New Test Suite 1');
remove(tsDel);

% Assign the system under test to the test case
setProperty(tc, 'Model', 'sldemo_autotrans');

% Turn on coverage settings at test-file level
cov = getCoverageSettings(tf);
cov.RecordCoverage = true;

% Enable MCDC and signal range coverage metrics
cov.MetricSettings = 'mr';

% Run the test case and return an object with results data
ro = run(tf);

% Get the coverage results
tfr = getTestFileResults(ro);
tsr = getTestSuiteResults(tfr);
tcs = getTestCaseResults(tsr);
cr = getCoverageResults(tcs);

% Open the Test Manager to view results
sltest.testmanager.view;
```

In the **Results and Artifacts** pane of the Test Manager, you can view the coverage results in the test case result.

## Create and Run Test Case Iterations

This example shows how to create test iterations. You can create table iterations programmatically that appear in the **Iterations** section of a test case. The example creates a simulation test case and assigns a Signal Builder group for each iteration.

```
% Create test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('Iterations Test File');
ts = getTestSuites(tf);
tc = createTestCase(ts, 'simulation', 'Simulation Iterations');

% Specify model as system under test
setProperty(tc, 'Model', 'sldemo_autotrans');

% Set up table iteration
% Create iteration object
testItr1 = sltestiteration;
% Set iteration settings
setTestParam(testItr1, 'SignalBuilderGroup', 'Passing Maneuver');
% Add the iteration to test case
addIteration(tc, testItr1);

% Set up another table iteration
% Create iteration object
testItr2 = sltestiteration;
% Set iteration settings
setTestParam(testItr2, 'SignalBuilderGroup', 'Coasting');
% Add the iteration to test case
addIteration(tc, testItr2);

% Run test case that contains iterations
results = run(tc);

% Get iteration results
```

```
tcResults = getTestCaseResults(results);  
iterResults = getIterationResults(tcResults);
```

## **See Also**

`sltest.testmanager.report`

## Run Combinations of Tests Using Iterations

<b>In this section...</b>
---------------------------

“Create Table Iterations” on page 6-70
--

“Create Scripted Iterations” on page 6-73
---

“Capture Baseline Data from Iterations” on page 6-76
--

“Sweep Through a Set of Parameters” on page 6-79
--

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

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

### Create Table Iterations

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

- 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 list.

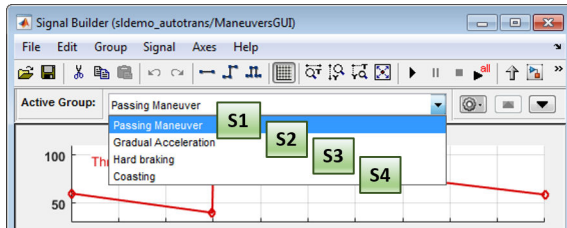


TABLE ITERATIONS

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

Auto Generate + Add Delete

Automatically 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. 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.



PARAMETER OVERRIDES

PARAMETER SET / WORKSPACE VARIABLE	_OVERRIDE VALUE	SOURCE	MODEL ELEMENT
Parameter Set 1	P1		
Parameter Set 2	P2		
Parameter Set 3	P3		

ITERATIONS

TABLE ITERATIONS

NAME	DESCRIPTION	REQUIREMENTS	SIGNAL BUILDER GROUP	PARAMETER SET
Iteration1	None	None	Passing Maneuver	Parameter Set 1
Iteration2	None	None	Gradual Acceleration	Parameter Set 2
Iteration3	None	None	Hard braking	Parameter Set 3

Auto Generate + Add Refresh Delete

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

**View Table Iterations**

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

**Generate Table Iterations**

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

When you automatically generate iterations, you can specify the naming rule for each iteration. Click **Auto Generate** and select the settings to use as the basis for the iterations. You can then use a rule to name the iterations to help to identify them. In the **Iteration naming rule** box, enter the rule using:

- The name of each setting you want to use in the name, with spaces removed
- An underscore or space to separate each setting

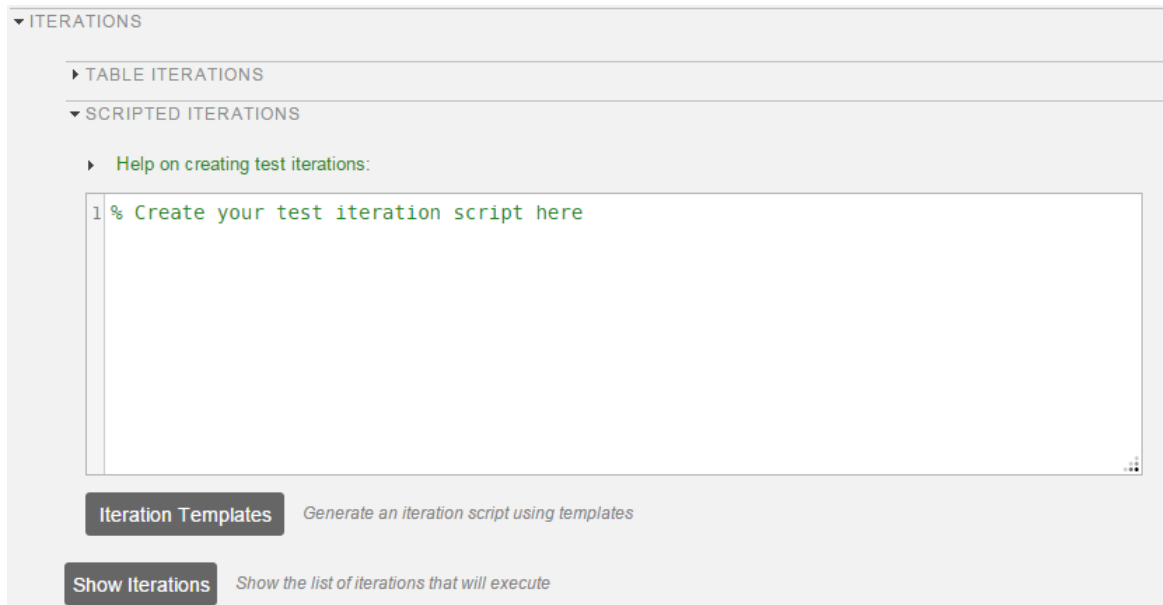
For example, if you want to include the name of the parameter set, configuration set, and baseline file name, enter `ParameterSet_ConfigurationSet_Baseline`. When you generate the iterations, the corresponding names of the test case replace each part of the rule.

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 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.

Section Option	Purpose
External Input	Applies to the <b>Inputs</b> section of a simulation, baseline, or equivalence test case, for the specified <b>External Inputs</b> 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 set is used to generate an iteration.
Simulation 1 or 2	Applies only to equivalence test case types. At the top of the Auto Generate Reports dialog box, there is a menu for <b>Simulation 1</b> or <b>Simulation 2</b> . These sections correspond to the two simulation sections within the equivalence test case.

## Create Scripted Iterations

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



## Iteration Script Components

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

```
%% Iterate Using a Signal Builder Group
```

```
% Set up a new iteration object
testItr = sltestiteration;
```

```
% Set iteration setting using Signal Builder group
setTestParam(testItr, 'SignalBuilderGroup', sltest_signalBuilderGroups{1});
```

```
% Add the iteration to run in this test case
% The predefined sltest_testCase variable is used here
addIteration(sltest_testCase, testItr);
```

For more information about the test iteration class, see `sltest.testmanager.TestIteration`. In practice, you iterate over numerous

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

```
%% Iterate Over All Signal Builder Groups

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

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

    % Set iteration settings
    setTestParam(testItr, 'SignalBuilderGroup', sltest_signalBuilderGroups{k});

    % Add the iteration to run in this test case
    % You can pass in an optional iteration name
    addIteration(sltest_testCase, testItr);
end
```

### Predefined Variables

You can use predefined variables to write iterations scripts. To see the list of predefined variables in the Test Manager, expand the **Help on creating test iterations** section. You write the iterations script in the script box within the **Scripted Iterations** section. The script box is a functional workspace, which means the MATLAB base workspace cannot access information from the script box. If you define variables in the script box, then other workspaces cannot use the variable.

The predefined variables are:

- `sltest_bdroot` — Model simulated by the test case, defined as a string
- `sltest_sut` — The System Under Test, defined as a string
- `sltest_isharness` — true if `sltest_bdroot` is a harness model, defined as a logical
- `sltest_externalInputs` — Name of external inputs, defined as a cell array of strings
- `sltest_parameterSets` — Name of parameter override sets, defined as a cell array of strings
- `sltest_configSets` — Name of configuration settings, defined as a cell array of strings

- `sltest_tableIterations` — Iteration objects created in the iterations table, defined as a cell array of `sltest.testmanager.TestIteration` objects
- `sltest_testCase` — Current test case object, defined as an `sltest.testmanager.TestCase` object

### Scripted Iteration Templates

You can quickly generate iterations for your test case using templates for Signal Builder groups, parameter sets, external inputs, configuration sets, and baseline sets, if you are using a baseline test case. Scripted iteration templates follow lockstep ordering and pairing of test settings. For more information about lockstep ordering, see “Create Table Iterations” on page 6-70.

For example, if you want to run all signal builder groups in a scripted iteration:

- 1 Click **Iteration Templates**.
- 2 Select the test case settings you want to iterate through. Click **OK**.

The script is generated and added to the script box below any existing scripts.

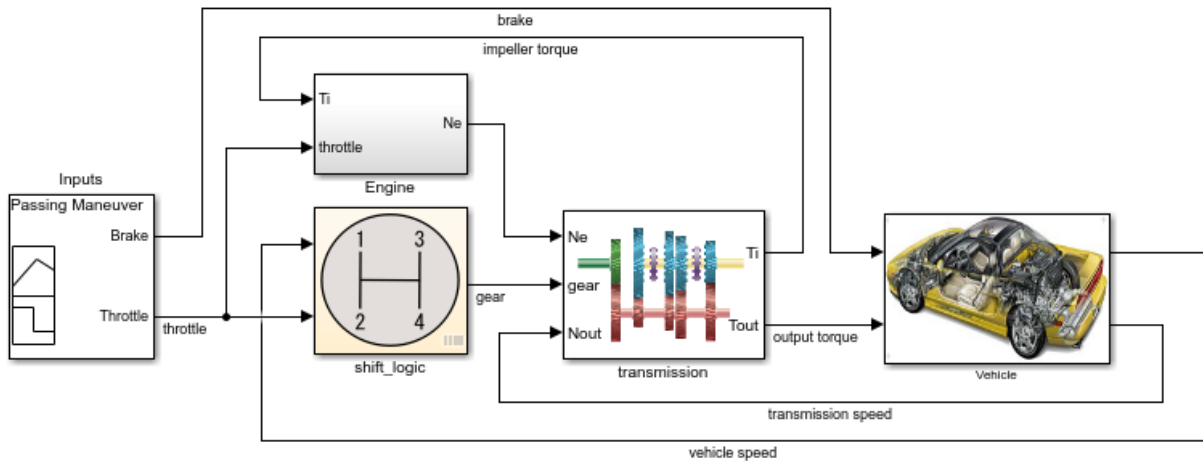
- 3 To generate a table that gives a preview of the iterations that execute when you run the test case, click **Show Iterations**.

### Capture Baseline Data from Iterations

This example shows how to create a baseline test by capturing data from a test case with table iterations. You create the iterations from Signal Builder groups in the model. Before running the example, navigate to a writable folder on the MATLAB® path.

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

```
Model = 'sltestCar';  
open_system(fullfile(matlabroot, 'examples', 'simulinktest', Model));
```

Simulink® Test™ model `sltestCar`

Copyright 1997-2017 The MathWorks, Inc.

2. Create a test file that contains iterations, and open the Test Manager. At the command line, enter

```
tf = sltest.testmanager.TestFile('IterationBaselineTest');
sltest.testmanager.load(tf.Name);
sltest.testmanager.view;
```

3. In the Test Manager, right-click the test case and select **Rename**. Rename the test case **Baseline Test**.

4. In the **System Under Test** section, for **Model**, enter `sltestCar`.

5. Select the signals for the baseline data:

- 1 In the **Simulation Outputs** section, click **Add**. The Signal Selection dialog box appears.
- 2 In the model canvas, select the `output torque` and `vehicle speed` signals. The signals appear in the Signal Selection dialog box.
- 3 In the dialog box, select both signals and click **Add**.
- 4 The signals appear in the **Logged Signals** table.

6. Add iterations for the test case:

- 1 Expand the **Iterations** section of the test case.
- 2 Expand the **Table Iterations** section and click **Auto Generate**.
- 3 In the dialog box, select **Signal Builder Group**. Click **OK**.
- 4 The table lists the iterations corresponding to the four Signal Builder groups.

7. Capture baseline data for the iterations:

- 1 In the **Baseline Criteria** section, click the arrow next to **Capture**, and select **Capture for Iterations**.
- 2 Specify a location for the baseline data files.
- 3 Click **Create**.

The model simulates for all Signal Builder groups. The baseline data for `output_torque` and `vehicle_speed` are captured in four MAT files. Also, each baseline data set is added to its corresponding iterations in the table.



▼ BASELINE CRITERIA\* ?

Include baseline data in test result

SIGNAL NAME	ABS TOL	REL TOL	LD TOL	LG TOL	
▶ <input type="checkbox"/> baseline1.mat	0	0.00%	0	0	
▶ <input type="checkbox"/> baseline2.mat	0	0.00%	0	0	
▶ <input type="checkbox"/> baseline3.mat	0	0.00%	0	0	
▶ <input checked="" type="checkbox"/> baseline4.mat	0	0.00%	0	0	

+ Add... Capture... Refresh Visualize Delete

▼ ITERATIONS\* ?

▼ TABLE ITERATIONS\* ?

<input checked="" type="checkbox"/> NAME	SIGNAL BUILDER GROUP	PARAMETER SET	EXTERNAL INPUT	BASELINE	
<input checked="" type="checkbox"/> Iteration1	Passing Maneuver	[Default] None	[Default] None	baseline1.mat	▲
<input checked="" type="checkbox"/> Iteration2	Gradual Acceleration	[Default] None	[Default] None	baseline2.mat	
<input checked="" type="checkbox"/> Iteration3	Hard braking	[Default] None	[Default] None	baseline3.mat	
<input checked="" type="checkbox"/> Iteration4	Coasting	[Default] None	[Default] None	baseline4.mat	▼

Auto Generate + Add Delete

## 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 6-63

## Collect Coverage in Tests

### In this section...

“Enable and Collect Coverage for a Test File” on page 6-81

“Considerations for Collecting Coverage in Test Harnesses” on page 6-84

If you use Simulink Coverage to generate model and code coverage, then you can collect coverage metrics when you run your test cases. For test cases with coverage collection turned on, Test Manager includes the coverage of each metric you specify to collect in the results.

To test a model for coverage, turn on coverage collection on the test file and specify the metrics you want to collect. Test suites and test cases inherit the settings from the test file. After coverage is turned on at the file level, you can turn it off and on for each test suite or test case.









### Enable and Collect Coverage for a Test File

Enable coverage collection, view coverage results in Test Manager, and trace coverage results from Test Manager to the model. The model `sldemo_autotrans` specifies coverage.

For information about types of model coverage, see “Types of Model Coverage” (Simulink Coverage).

















- 1 Create a test file and set up the test case for your model.
- 2 In the test file settings, under **Coverage Settings**, select **Record coverage for system under test**. You can also specify whether to collect coverage for referenced models.
- 3 Select the coverage metrics that you want to collect.
- 4 Run the test file.
- 5 To view the coverage results, select the test case result in the **Results and Artifacts** pane and expand the **Coverage Results** section.

COVERAGE RESULTS

















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

- 6 If your test file or test suite collected coverage metrics for more than one model, you can view all the coverage metrics in one place. Select the result and expand the **Aggregated Coverage Results** section.

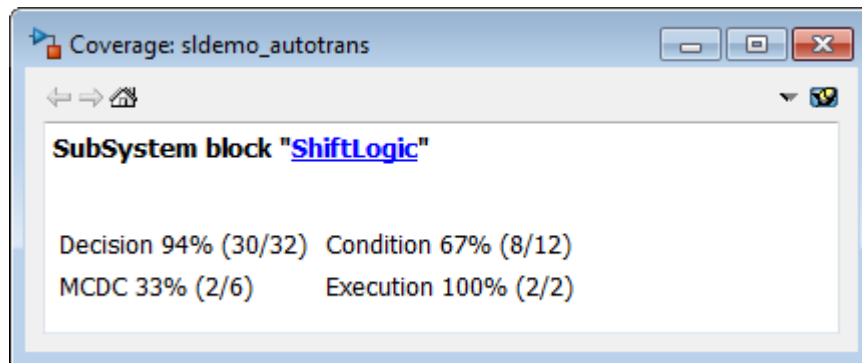
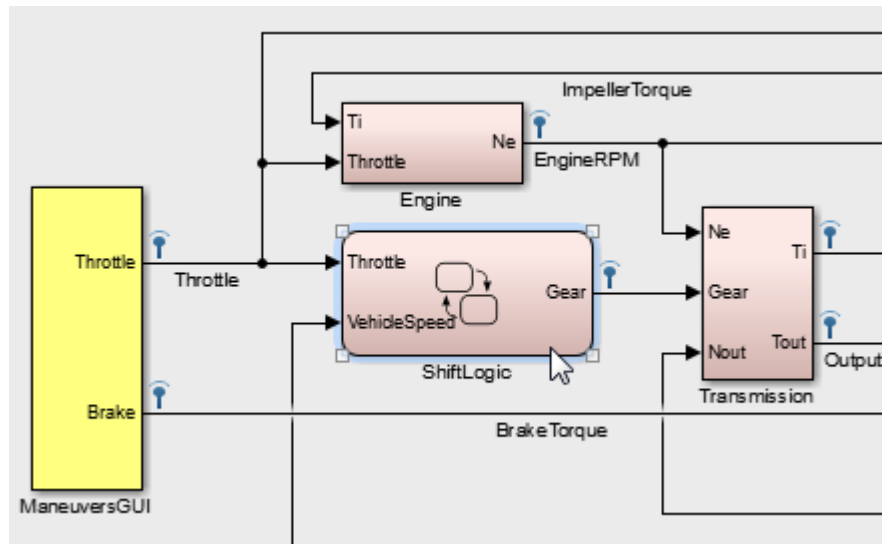
AGGREGATED COVERAGE RESULTS

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

- 7 To trace the coverage results to the model, click the model name in the coverage results table.

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

In the model, select model elements to see the coverage data.



- 8 To create a report of the coverage for a model, click the arrow under the **Report** column in the coverage results.

▼ AGGREGATED COVERAGE RESULTS

ANALYZED MODEL	REPORT	COMP.	D1
sf_aircraft		428	47%
sldemo_autotrans		24	94%

---

**Tip** To see aggregated results from different test files, in the **Results and Artifacts** pane, select the test file results whose coverage results you want to see in the same results file. From the context menu, select **Merge Coverage Results**. A results file that contains the combined coverage results appears in the list.

---

### Considerations for Collecting Coverage in Test Harnesses

Loading coverage results to a model, or aggregating coverage results across models, requires a model consistent with the coverage results. Therefore, to perform aggregated coverage collection, it is recommended that you use test harnesses configured to automatically synchronize the component under test. Set **SynchronizationMode** to `Synchronize on harness open and close`. For more information, see “Synchronize Changes Between Test Harness and Model” on page 2-57.

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

- If the test harness is configured to synchronize the component under test when you open or close the harness, coverage results from the test harness are associated with the main model. When you close the test harness, the coverage results remain active in memory. You can aggregate coverage with additional results collected from the main model or another synchronized test harness.
- If the test harness is configured to only synchronize the component under test when you manually push or rebuild, the coverage results are associated with the test harness.
  - When you close the test harness, the coverage results are removed from memory.
  - If the component under test design differs between test harness and main model, you cannot aggregate coverage results.
  - You can aggregate coverage results with the main model if the component under test design does not differ, but you must manually load the coverage results into the main model. See the function `cvload`.

### See Also

`sltest.testmanager.CoverageSettings`

## **Related Examples**

- “Perform Functional Testing and Analyze Test Coverage” on page 9-11
- “Automate Tests Programmatically” on page 6-63
- “Specify Coverage Options” (Simulink Coverage)

## Run Tests Using Parallel Execution

### In this section...

“Use Parallel Execution” on page 6-86

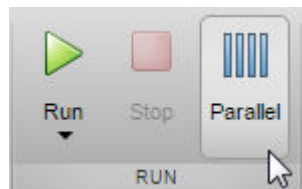
“When Do Tests Benefit from Using Parallel Execution?” on page 6-87

If you have a license to Parallel Computing Toolbox, then you can execute tests in parallel using a parallel pool (parpool). Running tests in parallel can speed up execution and decrease the amount of time it takes to get test results.

### Use Parallel Execution

To run a test file using parallel execution:

- 1 The Test Manager uses the default Parallel Computing Toolbox cluster. For information about where to specify or change the cluster, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox).
- 2 On the Test Manager toolstrip, click the **Parallel** button.



- 3 Run a test file. The test file executes using parallel pool.
- 4 To turn off parallel execution, click the **Parallel** button to toggle it off.

Starting a parallel pool can take time, which would slow down test execution. To reduce time:

- Make sure that the parallel pool is already running before you run a test. By default, the parallel pool shuts down after being idle for a specified number of minutes. To change the setting, see “Specify Your Parallel Preferences” (Parallel Computing Toolbox).
- Load Simulink on all the parallel pool workers.



## When Do Tests Benefit from Using Parallel Execution?

In general, parallel execution can help reduce test execution time if you have

- A complex Simulink model that takes a long time to simulate.
- Numerous long-running tests, such as iterations.

## See Also

`sltest.testmanager.run`

## Related Examples

- “Clusters and Clouds” (Parallel Computing Toolbox)

## Apply Tolerances to Test Criteria

### In this section...

“Modify Criteria Tolerances” on page 6-88

“Change Leading Tolerance in a Baseline Comparison Test” on page 6-88

You can specify tolerances that reflect the acceptable difference between simulation results and baseline data, or between simulation results and another simulation. Specify tolerances in the **Baseline Criteria** or **Equivalence Criteria** sections of test cases. You can specify relative, absolute, leading, and lagging tolerances. Relative and absolute tolerances correspond to signal values. Leading and lagging tolerances correspond to signal timing.

To learn about how tolerances are calculated, see “How the Simulation Data Inspector Compares Data” (Simulink).

### Modify Criteria Tolerances

To modify a tolerance, select the signal name in the criteria table, double-click the tolerance value, and enter a new value.

SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
▼ <input checked="" type="checkbox"/> My_mat_base.mat	0	0.00%	0	0
<input checked="" type="checkbox"/> Ww	0	0.00%	0	0
<input checked="" type="checkbox"/> Vs	0	0.00%	0	0
<input checked="" type="checkbox"/> Sd	0	0.00%	0	0
<input checked="" type="checkbox"/> slp	0	0.00%	0	0

If you modify a tolerance after you run a test case, rerun the test case to apply the new tolerance value to the simulation results.

### Change Leading Tolerance in a Baseline Comparison Test

This example shows how to change a tolerance in the simulation results, allowing you to preview the effect of a tolerance change before you change the test case. The example uses a leading tolerance. A solver change causes a difference between simulation output and baseline data.

## Generate the Baseline

Generate the baseline for the `sf_car` model, which uses the `ode-5` solver.

- 1 Open the model `sf_car`.
- 2 Open the Test Manager and create a test file named `Solver Compare`. In the test case, set the system under test to `sf_car`.
- 3 Select the signal to log. Under **Simulation Outputs**, click **Add**. In the model, select the `shift_logic` output signal. In the Signal Selection dialog box, select `shift_logic` and click **Add**.
- 4 Save the baseline data. Under **Baseline Criteria**, click **Capture**. Set the file format to MAT. Name the file `solver_baseline` and click **Capture**.

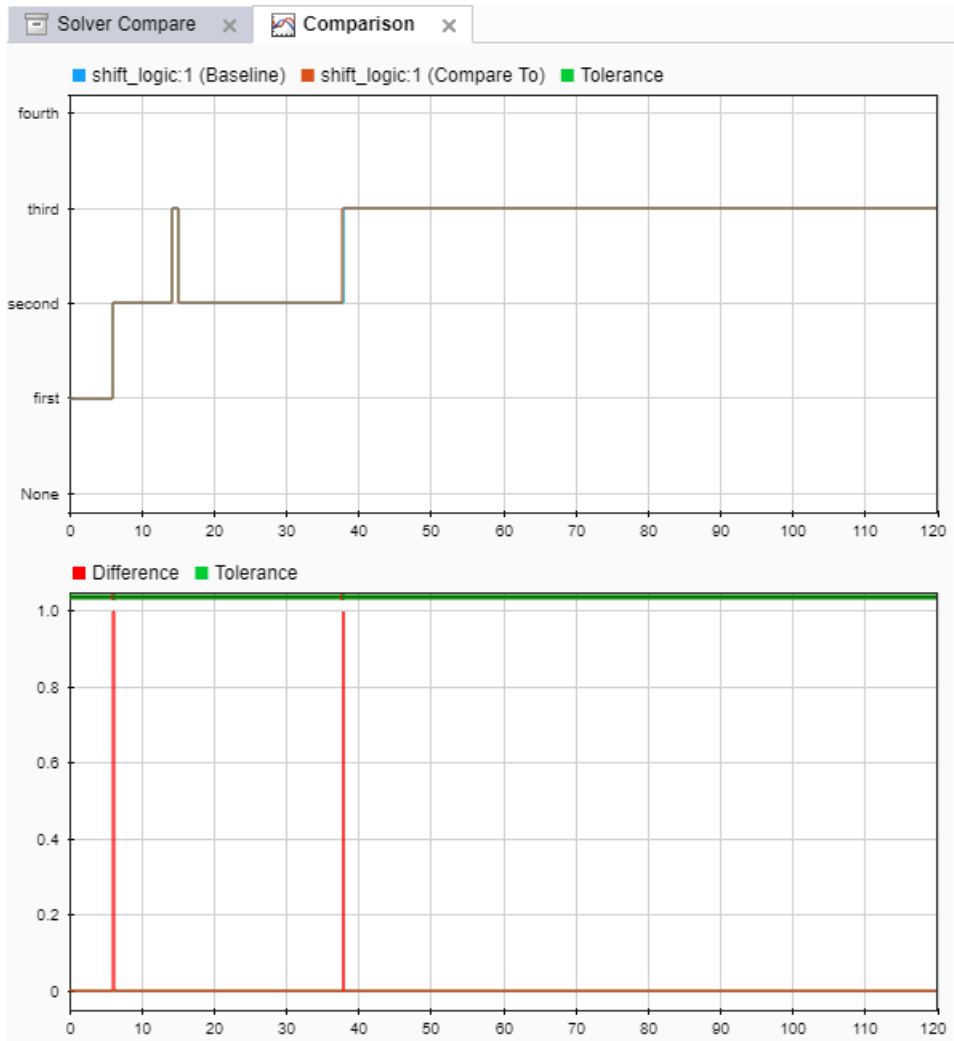
The model simulates, and the baseline criteria appear in the table.

SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
▼ <input checked="" type="checkbox"/> solver_baseline.mat	0	0.00%	0	0
<input checked="" type="checkbox"/> shift_logic:1	0	0.00%	0	0

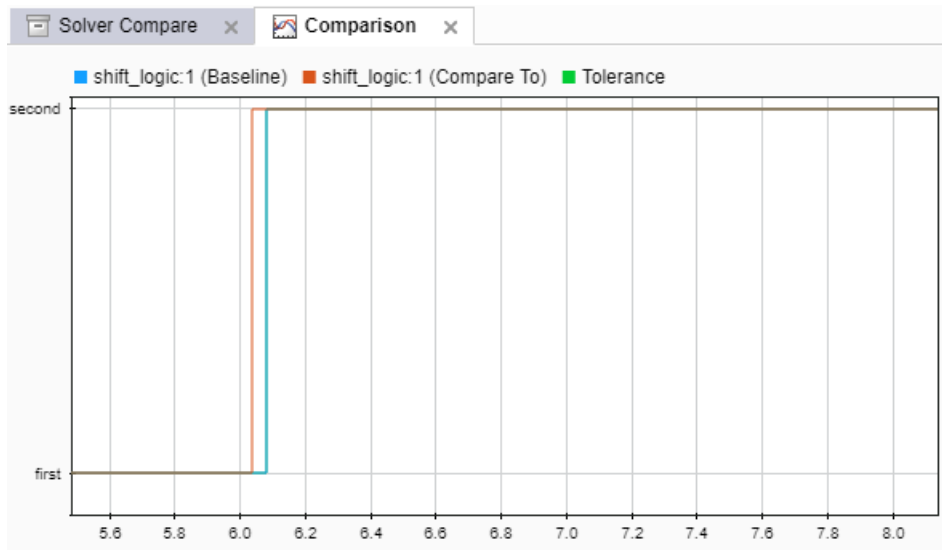
## Change the Solver and Run the Test Case

- 1 In the model, change the solver to `ode1`.
- 2 In the Test Manager, run the `Solver Compare` test file. The results fail.
- 3 Expand the test results. Under **Baseline Criteria Result**, select the `shift_logic` signal.

The **Comparison** tab shows the simulation output, baseline data, and difference.



4 Zoom in on the first signal difference. The comparison signal leads the baseline data.



### Preview and Set a Leading Tolerance Value

Suppose that a change in the gear signal output would be acceptable within  $0.04$  s ahead of the baseline data. In the Test Manager, set a leading tolerance value. You can preview how the tolerance value affects the test to see if the test passes with the specified tolerance. Then, set the tolerance in the test case and rerun the test.

- 1 With the result signal selected, in the property box, set **Leading Tolerance** to  $.04$ . When you change this value, the result status changes to **pass**.

PROPERTY	VALUE
Name	<input checked="" type="checkbox"/> shift_logic:1
Status	<span style="color: green;">✔</span>
Absolute Tolerance	0
Relative Tolerance	0.00%
Leading Tolerance	<span style="border: 1px solid blue; padding: 2px;">0.04</span>
Lagging Tolerance	0
Block Path	sf_car/shift_logic

- 2 Enter the tolerance value in the test case. In the **Test Browser**, under **Baseline Criteria**, change the **Leading Tol** value to  $.04$ .

SIGNAL NAME	ABS TOL	REL TOL	LEADING TOL	LAGGING TOL
▼ <input checked="" type="checkbox"/> solver_baseline.mat	0	0.00%	0.04	0
<input checked="" type="checkbox"/> shift_logic:1	0	0.00%	0.04	0

- 3 Rerun the test and save the new result.
- 4 To store the tolerance value and the passed test with the test file, save the test file.

### See Also

sltest.testmanager.BaselineCriteria |  
sltest.testmanager.SignalCriteria

### Related Examples

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

## Test Manager Limitations

### In this section...

- “Simulation Mode” on page 6-93
- “Callback Scripts” on page 6-93
- “Protected Models” on page 6-93
- “Parameter Overrides” on page 6-94
- “Breakpoints” on page 6-94
- “Highlight in Model” on page 6-94

### Simulation Mode

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

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

### Callback Scripts

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

- The Test Manager cannot stop the execution of an infinite loop inside a callback script. To stop execution of an infinite loop from a callback script, press **Ctrl+C** at the MATLAB command prompt.
- `sltest.testmanager` functions are not supported.

### Protected Models

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

### Parameter Overrides

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

### Breakpoints

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

### Highlight in Model

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



## Test Sections

### In this section...

“Select Releases for Testing” on page 6-95  
“Set Preferences to Display Test Sections” on page 6-96  
“Select releases for simulation” on page 6-96  
“Tags” on page 6-96  
“Description” on page 6-96  
“Requirements” on page 6-97  
“System Under Test” on page 6-97  
“Parameter Overrides” on page 6-98  
“Callbacks” on page 6-98  
“Inputs” on page 6-100  
“Simulation Outputs” on page 6-100  
“Configuration Setting Overrides” on page 6-101  
“Simulation 1 and Simulation 2” on page 6-102  
“Equivalence Criteria” on page 6-102  
“Baseline Criteria” on page 6-102  
“Custom Criteria” on page 6-104  
“Iterations” on page 6-104  
“Coverage Settings” on page 6-105  
“Test File Options” on page 6-105

To view or edit the test sections, select a test file, suite, or case in the **Test Browser** pane. For information on the types of test cases, see “Introduction to Test Manager” on page 5-2.

### Select Releases for Testing

You can select MATLAB releases installed on your system to create and run tests in. Use this preference to specify the MATLAB installations that you want to make available for testing with Test Manager. You can use releases from R2011b forward. The releases you

add become available to select from the **Select releases for simulation** list when you design the test.

You can add releases to the list and delete them. You cannot delete the release you started MATLAB in.

To add a release, click **Add**, navigate to the location of the MATLAB installation you want to add, and click **OK**.

For more information, see “Run Tests in Multiple Releases” on page 6-52.

### Set Preferences to Display Test Sections

To simplify the Test Manager layout, you can select the sections of the test case, test suite, or test file that appear in the Test Manager. Test case sections that were modified appear in the Test Manager, regardless of the preference setting.

- 1 In the toolstrip, click **Preferences**.
- 2 Select the **Test File**, **Test Suite**, or **Test Case** tab.
- 3 Select sections to show, or clear sections to hide. To show only sections where settings are set, clear all selections in the **Preferences** dialog box.
- 4 Click **OK**.

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

### Select releases for simulation

Select the releases that you want available for running test cases. Build the list of releases using the **Release** pane in the Test Manager Preferences dialog box. For more information, see “Run Tests in Multiple Releases” on page 6-52.

### Tags

Tag your tests with useful categorizations, such as `safety`, `logged-data`, or `burn-in`. Filter tests using these tags when executing tests or viewing results. See “Filter Test Execution and Results” on page 6-134.

### Description

In this section, add descriptive text to your test case, test suite, or test file.


## Requirements

If you have a Simulink Requirements license, you can establish traceability by linking your test cases to requirements. For more information, see “Link to Test Cases from Requirements” (Simulink Requirements).

To link a test case, test suite, or test file to a requirement:

- 1 Open the Requirements Editor. In the Simulink menu, select **Analysis > Requirements > Requirements Editor**.
- 2 Highlight a requirement.
- 3 In the Test Manager, in the **Requirements** section, click the arrow next to the **Add** button and select **Link to Selected Requirement**.
- 4 The requirement link appears in the **Requirements** list.


## System Under Test

Specify the model you want to test in the **System Under Test** section. To use an open model in the currently active Simulink window, click the **Use current model** button .


---

**Note** The model must be available on the path to run the test case. You can add the model's containing folder to the path using the preload callback. See “Callbacks” on page 6-98.

---

Specifying a new model in the **System Under Test** section can cause the model information to be out of date. To update the model test harnesses, Signal Builder groups, and available configuration sets, click the **Refresh** button .

### Test Harness

If you have a test harness in your system under test, then you can select the test harness to use for the test case. If you have added or removed test harnesses in the model, click the **Refresh** button  to view the updated test harness list.

For more information about using test harnesses, see “Refine, Test, and Debug a Subsystem” on page 2-24.

### 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

In this section, you can specify parameter values in the test case to override the parameter values in the model workspace, data dictionary, or base workspace. Parameters are grouped into sets. You can turn parameter sets and individual parameter overrides on or off by using the check box next to the set or parameter.

To add a parameter override:

- 1 Click **Add**.

A dialog box opens with a list of parameters. If the list of parameters is not current, click the **Refresh** button  in the dialog box.

- 2 Select the parameter you want to override.
- 3 To add the parameter to the parameter set, click **OK**.
- 4 Enter the override value in the parameter **Override Value** column.

To restore the default value of a parameter, clear the value in the **Override Value** column and press **Enter**.

You can also add a set of parameter overrides from a MAT-file. Click the **Add** arrow and select **Add File** to create a parameter set from a MAT-file.

For an example that uses parameter overrides, see “Overriding Model Parameters in a Test Case”.

### Callbacks

#### Test-File Level Callbacks

Two callback scripts are available in each test suite that execute at different times during a test:

- Setup runs before test file executes.

- Cleanup runs after test file executes.

### Test-Suite Level Callbacks


Two callback scripts are available in each test suite that execute at different times during a test:

- Setup runs before the test suite executes.
- Cleanup runs after the test suite executes.

### Test-Case Level Callbacks

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

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

To run a single callback script, click the **Run** button  above the corresponding script.

See “Test Manager Limitations” on page 6-93 for the limitations of callback scripts in test cases. For information on Simulink model callbacks, see “Model Callbacks” (Simulink).

You can use these predefined variables in the test case callbacks:

- `sltest_bdroot` available in **Post-Load**: The model simulated by the test case. The model can be a harness model.
- `sltest_sut` available in **Post-Load**: The system under test. For a harness, it is the component under test.
- `sltest_isharness` available in **Post-Load**: Returns true if `sltest_bdroot` is a harness model.
- `sltest_simout` available in **Cleanup**: Simulation output produced by simulation.
- `sltest_iterationName` available in **Pre-Load**, **Post-Load**, and **Cleanup**: Name of the currently executing test iteration.

### Inputs

For test inputs, you can use inputs from signal builder groups in the model, or you can use external inputs from MAT-files or Microsoft Excel files.

- To use inputs from a Signal Builder block group in the model or test harness, select the **Signal Builder Group** check box, and then select the group from the list.
- To select an external input set in the **External Inputs** table to run when the test case executes, click **Add**. Select a MAT-file or Microsoft Excel file to import as inputs to your test case. Select the file you want to use from the table.

For more information on using external files as inputs, see “Use External Inputs in Test Cases” on page 6-42. For information about the file format for Microsoft Excel files in Test Manager, see “Specify Microsoft Excel File Format for Signal Data” on page 6-25.

You can also use Test Manager to create input file templates as MAT-files or Excel files and enter the data in them. See “Create Data Files to Use as Test Inputs” on page 6-49.

### Edit Input Data Files in Test Manager

From the Test Manager, you can edit your input data files.

To edit a file, select the file and click **Edit**. You can then edit the data in the signal editor for MAT-files or Microsoft Excel for Excel files.

To learn about the syntax for Excel files, see “Specify Microsoft Excel File Format for Signal Data” on page 6-25.

### Simulation Outputs

Use the **Simulation Outputs** section to add signal outputs to your test results. Signals logged in your model or test harness can appear in the results after you add them as simulation outputs. You can then plot them. Add individual signals to log and plot or add a signal set.

- 1 Under **Simulation Outputs**, click **Add**.
- 2 In the system under test, select the signals you want to log. Adding signals to log in the test case does not alter the model or test harness.
- 3 In the **Signal Selection** dialog box, select the check box next to the signals whose output you want to capture, and click **Add**.

- To add a signal set, click the **Add** arrow and select **Signal Set**.
- 4 To specify a specific plot for a signal, enter a plot index number. By default, the signals appear on one plot.

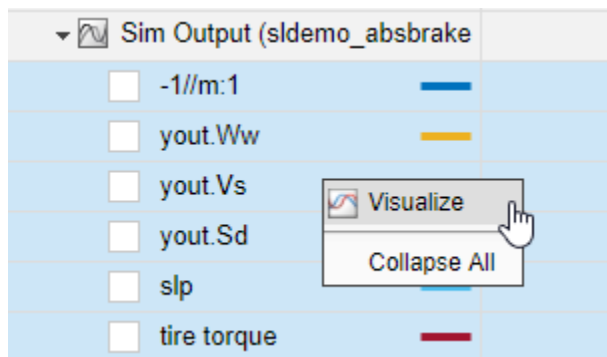
You can specify to display the plot immediately after running the test by selecting the **Plot signals on the specified plots after simulation** check box.

After you run the test, the logged signals appear in the test case results under **Sim Output**. Select each signal to display it on the plot. If you specified a plot index, the signal appears in the plot number you specified.

---

**Tip** To plot multiple signals from test results, select them and select **Visualize** from the context menu.

---




---

**Note** When you run a test multiple times, by default the new signals are added to the plot from previous test runs. To instead overwrite the plots with only the new results, right-click **Sim Output** and select **Plot Signals > Overwrite**.

---

## Configuration Setting Overrides

In the test case, you can specify configuration settings that differ from the settings in the model. Setting the configuration settings in the test case enables you to try different configurations without modifying your model.

### Simulation 1 and Simulation 2

These sections appear in equivalence test cases. Use them to specify the details about the simulations that you want to compare. Enter the system under test, the test harness if applicable, and simulation setting overrides under **Simulation 1**. You can then click **Copy settings from Simulation 1** under **Simulation 2** to use a starting point for your second set of simulation settings.

For the test to pass, Simulation 1 and Simulation 2 must log the same signals.

Use these sections with the **Equivalence Criteria** section to define the premise of your test case. For an example of an equivalence test, see “Test Two Simulations for Equivalence”.

### Equivalence Criteria

This section appears in equivalence test cases. The equivalence criteria is a set of signal data to compare in Simulation 1 and Simulation 2. Specify tolerances to regulate pass-fail criteria of the test. You can specify absolute, relative, leading, and lagging tolerances for the signals.

To specify tolerances, first click **Capture** to run the system under test in Simulation 1 and add signals marked for logging to the table. Specify the tolerances in the table.

After you capture the signals, you can select signals from the table to narrow your results. If you do not select signals under **Equivalence Criteria**, running the test case compares all the logged signals in Simulation 1 and Simulation 2.

For an example of an equivalence test case, see “Test Two Simulations for Equivalence”.

### Baseline Criteria

The **Baseline Criteria** section appears in baseline test cases. When a baseline test case executes, Test Manager captures signal data from signals in the model marked for logging and compares them to the baseline data.

#### Capture Baseline Criteria

To capture logged signal data from the system under test to use as the baseline criteria, click **Capture**. Then follow the prompts in the Capture Baseline dialog box. Capturing the



data compiles and simulates the system under test and stores the output from the logged signals to the baseline. For a baseline test example, see “Test Model Output Against a Baseline” on page 6-9.

You can save the signal data to a MAT-file or a Microsoft Excel file. To understand the format of the Excel file, see “Specify Microsoft Excel File Format for Signal Data” on page 6-25.

You can capture the baseline criteria using the current release for simulation or another release installed on your system. Add the releases you want to use in the Test Manager preferences. Then, select the releases you want available in your test case using the **Select releases for simulation** option in the test case. When you run the test, you can compare the baseline against the release you created the baseline in or against another release. For more information, see “Run Tests in Multiple Releases” on page 6-52.

When you select Excel as the output format, you can specify the sheet name to save the data to. If you use the same Excel file for input and output data, by default both sets of data appear in the same sheet.

If you are capturing the data to a file that already contains outputs, specify the sheet name to overwrite the output data only in that sheet of the file.

To save a baseline for each test case iteration in a separate sheet in the same file, select **Capture a baseline for each iterations**. This check box appears only if your test case already contains iterations. For more information iterations, see “Run Combinations of Tests Using Iterations” on page 6-70.

### Specify Tolerances

You can specify tolerances to determine the pass-fail criteria of the test case. You can specify absolute, relative, leading, and lagging tolerances for individual signals or the entire baseline criteria set.

After you capture the baseline, the baseline file and its signals appear in the table. In the table, you can set the tolerances for the signals. To see tolerances used in an example for baseline testing, see “Test Model Output Against a Baseline” on page 6-9.

### Add File as Baseline

By clicking **Add**, you can select an existing file as a baseline. You can add MAT-files and Microsoft Excel files as the baseline. Format Microsoft Excel files as described in “Specify Microsoft Excel File Format for Signal Data” on page 6-25.

### Update Signal Data in Baseline

You can edit the signal data in your baseline, for example, if your model changed and you expect different values. To open the signal editor or the Microsoft Excel file for editing, select the baseline file from the list and click **Edit**. See “Manually Update Signal Data in a Baseline” on page 6-60.

You can also update your baseline when you examine test failures in the data inspector view. See “Examine Test Failures and Modify Baselines” on page 6-57.

### Custom Criteria

This section includes an embedded MATLAB editor to define custom pass/fail criteria for your test. Select **function customCriteria(test)** to enable the criteria script in the editor. Custom criteria operate outside of model run time; the script evaluates after model simulation.

Common uses of custom criteria include verifying signal characteristics or verifying test conditions. MATLAB Unit Test qualifications provide a framework for verification criteria. For example, this custom criteria script gets the last value of the signal PhiRef and verifies that it equals 0:

```
% Get the last value of PhiRef from the dataset Signals_Req1_3
lastValue = test.sltest_simout.get('Signals_Req1_3').get('PhiRef').Values.Data(end);

% Verify that the last value equals 0
test.verifyEqual(lastValue,0);
```

See “Apply Custom Criteria to Test Cases” on page 6-111. For a list of MATLAB Unit Test qualifications, see “Types of Qualifications” (MATLAB).

You can also define plots in the **Custom Criteria** section. See “Create, Store, and Open MATLAB Figures” on page 6-122.

### Iterations

Use this test case section 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 Combinations of Tests Using Iterations” on page 6-70.

## Coverage Settings

Use this test section to configure coverage collection for test files, test suites, and test cases. For more information about collecting coverage in your test, see “Collect Coverage in Tests” on page 6-81.

## Test File Options

### Close open Figures at the end of execution

When your tests generate figures, select this option to clear the working environment of figures after the test execution completes.

### Store MATLAB figures

Select this option to store figures generated during the test with the test file. You can enter MATLAB code that creates figures and plots as a callback or in the test case **Custom Criteria** section. See “Create, Store, and Open MATLAB Figures” on page 6-122.

### Generate report after execution

Select **Generate report after execution** to create a report after the test executes. Selecting this option displays report options that you can set. The settings are saved with the test file.

For detailed reporting information, see “Export Test Results and Generate Reports” on page 7-9 and “Customize Test Reports” on page 7-15.

## See Also

`sltest.testmanager.getpref` | `sltest.testmanager.setpref`

## Test Models Using Inputs Generated by Simulink Design Verifier

In this section...
“Overall Workflow” on page 6-106
“Test Case Generation Example” on page 6-107

Using Simulink Design Verifier, you can generate test inputs that replicate design errors, achieve test objectives, or meet coverage criteria. Simulink Test can create test cases that use test inputs and expected outputs from Simulink Design Verifier.

### Overall Workflow

Test case generation follows this workflow.

- 1 Choose an existing Simulink Design Verifier results file, or generate new results by analyzing your model.
  - If you use an existing results file, you can load results by either:
    - Using the Simulink Test command `sltest.import.sldvData`.
    - Using Simulink Design Verifier menu items. In the model, select **Analysis > Design Verifier > Results > Load**. Select the MAT file with the analysis results.
  - If you run a model analysis, the Simulink Design Verifier Results Summary window appears after the analysis completes.
- 2 In the results summary window, click **Export test cases to Simulink Test**.
- 3 Enter the name of an existing or new test harness.
- 4 Select a test harness source for the generated test inputs. You can select
  - **Inport**: The inputs are contained in the Simulink Design Verifier data file and mapped to Inport blocks in the test harness. The mapping is shown in the **Inputs** section of the test case. Using the **Inport** option allows you to map other inputs to the test harness Inport blocks, which can be useful for running multiple test cases or iterations using the same test harness.

- **Signal Builder**: The inputs are contained in groups in a Signal Builder block inside the test harness. Using the **Signal Builder** option allows you to view the test inputs in the Signal Builder block editor.
- 5 Select a new or existing test file, and enter names for the test file and test case.
  - 6 Click OK to export the test cases to Simulink Test. The test files and test cases are updated in the Test Manager.

## Test Case Generation Example

This example shows how to generate test cases for a controller subsystem using Simulink Design Verifier, and export the test cases to a test file in Simulink Test. The example requires a Simulink Design Verifier license.

The model is a closed-loop heat pump system. The controller accepts the measured room temperature and set temperature inputs. The controller outputs a bus of three signals controlling the fan, heat pump, and the direction of the heat pump (heat or cool). The model contains a harness that tests heating and cooling scenarios.

- 1 Open the model.

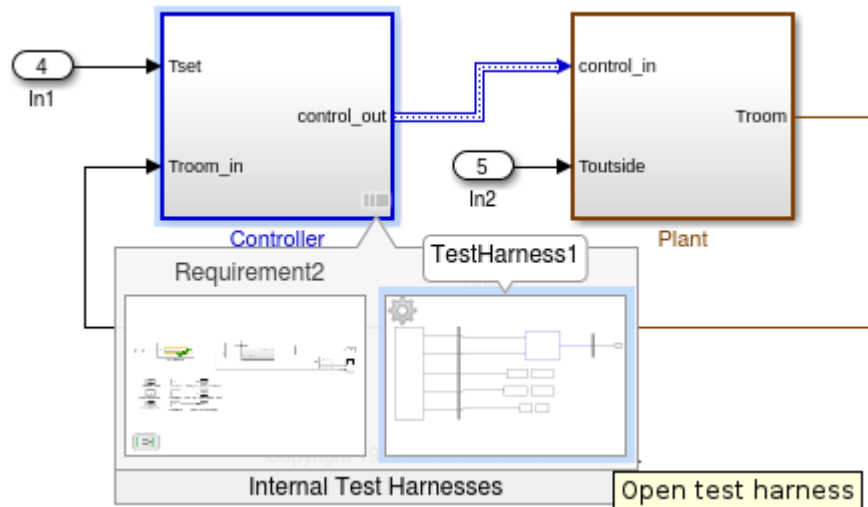
```
open_system(fullfile(docroot, 'toolbox', 'sltest', 'examples', ...  
'sltestTestCaseFromDVExample.slx'));
```

- 2 Set the current working folder to a writable folder.
- 3 In the model, generate tests for the **Controller** subsystem. Right-click the **Controller** block and select **Design Verifier > Generate Tests for Subsystem**.

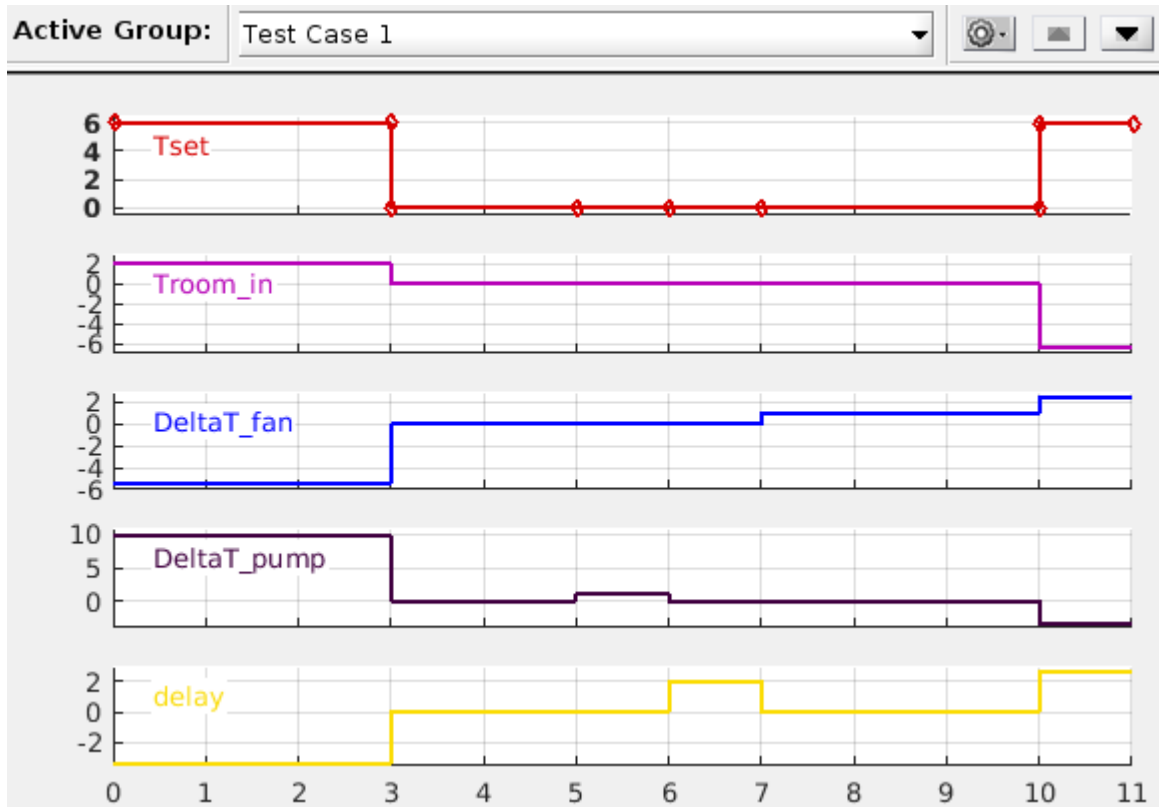
Simulink Design Verifier generates tests for the component.

- 4 In the results summary window, click **Export test cases to Simulink Test**.
- 5 In the Export Design Verifier Test Cases dialog box, enter:
  - Test Harness: TestHarness1
  - Harness Source: Signal Builder
  - Select **Use a new test file**
  - Test File: ./TestFile\_GeneratedTests.mldatx
  - Test Case: <Create a new test case>
- 6 Click **OK**.

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



- 7 Click the **TestHarness1** thumbnail to open the harness, and double-click the Signal Builder block source to see the generated inputs.



- 8 In the Test Manager, the new test case displays the system under test, and the test harness containing the generated inputs in the Signal Builder source. Expand the **Iterations** section to see the iterations corresponding to the signal builder groups.

The screenshot displays the Test Manager interface. On the left, the 'Test Browser' pane shows a tree view with 'TestFile\_GeneratedTests' expanded to 'New Test Suite 1', which contains 'New Test Case 1'. The main area shows the 'New Test Case 1' configuration. Under 'ITERATIONS\*', there is a 'TABLE ITERATIONS\*' section with a table of six test cases. Each row has a checked checkbox, a name, a signal builder group, a parameter set, an external input, and a logged signal set. All parameter sets, external inputs, and logged signal sets are set to '[Default] None'. At the bottom right of the table area, there are buttons for 'Auto Generate', '+ Add', and 'Delete'.

<input checked="" type="checkbox"/>	NAME	SIGNAL BUILDER GROUP	PARAMETER SET	EXTERNAL INPUT	LOGGED SIGNAL SET
<input checked="" type="checkbox"/>	Test Case 1	Test Case 1	[Default] None	[Default] None	[Default] None
<input checked="" type="checkbox"/>	Test Case 2	Test Case 2	[Default] None	[Default] None	[Default] None
<input checked="" type="checkbox"/>	Test Case 3	Test Case 3	[Default] None	[Default] None	[Default] None
<input checked="" type="checkbox"/>	Test Case 4	Test Case 4	[Default] None	[Default] None	[Default] None
<input checked="" type="checkbox"/>	Test Case 5	Test Case 5	[Default] None	[Default] None	[Default] None
<input checked="" type="checkbox"/>	Test Case 6	Test Case 6	[Default] None	[Default] None	[Default] None

### See Also

`sltest.import.sldvData`



## Apply Custom Criteria to Test Cases

### In this section...

“MATLAB Testing Framework” on page 6-111

“Define a Custom Criteria Script” on page 6-112

“Reuse Custom Criteria and Debug Using Breakpoints” on page 6-113

“Assess the Damping Ratio of a Flutter Suppression System” on page 6-115

“Custom Criteria Programmatic Interface Example” on page 6-120

Testing your model often requires assessing conditions that ensure a test is valid, in addition to verifying model behavior. MATLAB Unit Test provides a framework for such assessments. In Simulink Test, you can use the test case custom criteria to author specific assessments, and include MATLAB Unit Test qualifications in your script.

Custom criteria apply as post-simulation criteria to the simulation output. If you require run-time verifications, use a `verify()` statement in a Test Assessment or Test Sequence block. See “Run-Time Assessments” on page 3-49.

### MATLAB Testing Framework

A custom criteria script is a method of `test`, which is a `matlab.unittest` test case object. To enable the function, in the test case **Custom Criteria** section of the Test Manager, select **function customCriteria(test)**. Inside the function, enter the custom criteria script in the embedded MATLAB editor.

The embedded MATLAB editor lists properties of `test`. Create test assessments using MATLAB Unit Test qualifications. Custom criteria supports verification and assertion type qualifications. See “Types of Qualifications” (MATLAB). Verifications and assertions operate differently when custom criteria are evaluated:

- Verifications - Other assessments are evaluated when verifications fail. Diagnostics appear in the results. Use verifications for general assessments, such as checking simulation against expected outputs.

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

- Assertions - The custom criteria script stops evaluating when an assertion fails. Diagnostics appear in the results. Use assertions for conditions that render the criteria invalid.

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

## Define a Custom Criteria Script

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

- 1 Open the test file.

```
sltest.testmanager.load('AutopilotTestFile.mldatx')
sltest.testmanager.view
```

- 2 In the **Test Browser**, select **AutopilotTestFile > Basic Design Test Cases > Requirement 1.3 Test**. In the test case, expand the **Custom Criteria** section.
- 3 Enable the custom criteria script by selecting function `customCriteria(test)`.
- 4 In the embedded MATLAB editor, enter the following script. The script gets the final value of the signals Phi and APEng, and verifies that the final values equal 0.

```
% Get the last values
lastPhi = test.sltest_simout.get('Signals_Req1_3').get('Phi').Values.Data(end);
lastAPEng = test.sltest_simout.get('Signals_Req1_3').get('APEng').Values.Data(end);

% Verify the last values equal 0
test.verifyEqual(lastPhi,0,['Final Phi value: ',num2str(lastPhi),'.']);
test.verifyEqual(lastAPEng,false,['Final APEng value: ',num2str(lastAPEng),'.']);
```

- 5 Run the test case.
- 6 In the **Results and Artifacts** pane, expand the **Custom Criteria** Result. Both criteria pass.

▼ [icon] Requirement 1.3 Test	✖
▶ [icon] Verify Statements	✖
▶ [icon] Sim Output (RollAutopilotMdlRef :	
▼ [icon] Custom Criteria Result	✔
<input checked="" type="checkbox"/> Final Phi value: 0. { verifyEqual	✔
<input checked="" type="checkbox"/> Final APEng value: 0. { verifyEq	✔

## Reuse Custom Criteria and Debug Using Breakpoints

In addition to authoring criteria scripts in the embedded MATLAB editor, you can author custom criteria in a standalone function, and call the function from the test case. Using a standalone function allows you

- To reuse the custom criteria in multiple test cases.
- To set breakpoints in the criteria script for debugging.
- To investigate the simulation output using the command line.

In this example, you add a breakpoint to a custom criteria script. You run the test case, list the properties of the test object at the command line, and call the custom criteria from the test case.

### Call Custom Criteria Script from the Test Case

- 1 Navigate to the folder containing the criteria function.

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

- 2 Open the custom criteria script

```
open('sltestCheckFinalRollRefValues.m')
```

```
% This is a custom criteria function for a Simulink Test test case.
% The function gets the last values of Phi and APEng from the
% Requirements 1.3 test case in the test file AutopilotTestFile.
```

```
function sltestCheckFinalRollRefValues(test)
```

```
% Get the last values
```

```
lastPhi = test.sltest_simout.get('Signals_Req1_3').get('Phi').Values.Data(end)
lastAPEng = test.sltest_simout.get('Signals_Req1_3').get('APEng').Values.Data(end)
```

```
% Verify the last values equal 0
```

```
test.verifyEqual(lastPhi,0,['Final Phi value: ',num2str(lastPhi),'.']);
test.verifyEqual(lastAPEng,false,['Final APEng value: ',num2str(lastAPEng),'.']);
```

- 3 Open the test file

```
sltest.testmanager.load('AutopilotTestFile.mldatx')
sltest.testmanager.view
```

- 4 In the embedded MATLAB editor under **Custom Criteria**, enter the function call to the custom criteria:

```
sltestCheckFinalRollRefValues(test)
```

### Set Breakpoints and List test Properties

- 1 On line 8 of `sltestCheckFinalRollRefValues.m`, set a breakpoint by clicking the dash to the right of the line number.
- 2 In the Test Manager, run the test case.

The command window displays a debugging prompt.

- 3 Enter `test` at the command prompt to display the properties of the `STMCustomCriteria` object. The properties contain characteristics and simulation data output of the test case.

```
test =
```

```
STMCustomCriteria with properties:
```

```
    TestResult: [1x1 sltest.testmanager.TestCaseResult]
    sltest_simout: [1x1 Simulink.SimulationOutput]
    sltest_testCase: [1x1 sltest.testmanager.TestCase]
    sltest_bdroot: {'RollReference_Requirement1_3'}
    sltest_sut: {'RollAutopilotMdlRef/Roll Reference'}
    sltest_isharness: 1
    sltest_iterationName: ''
```

The property `sltest_simout` contains the simulation data. To view the data `PhiRef`, enter

```
test.sltest_simout.get('Signals_Req1_3').get('PhiRef')
```

```
ans =
```

```
Simulink.SimulationData.Signal
Package: Simulink.SimulationData
```

```
Properties:
struct with fields:
```

```
    Name: 'PhiRef'
    PropagatedName: ''
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
    PortType: 'outport'
    PortIndex: 1
    Values: [1x1 timeseries]
```

- 4 In the MATLAB editor, click **Continue** to continue running the custom criteria script.
- 5 In the **Results and Artifacts** pane, expand the **Custom Criteria** Result. Both criteria pass.
- 6 To reuse the script in another test case, call the function from the test case custom criteria.

## Assess the Damping Ratio of a Flutter Suppression System

Using a custom criteria script, verify that wing oscillations are damped in multiple altitude and airspeed conditions.

### The Simulation and Model

The model uses Simscape™ to simulate a Benchmark Active Controls Technology (BACT) / Pitch and Plunge Apparatus (PAPA) setup. It uses Aerospace Blockset™ to simulate aerodynamic forces on the wing.

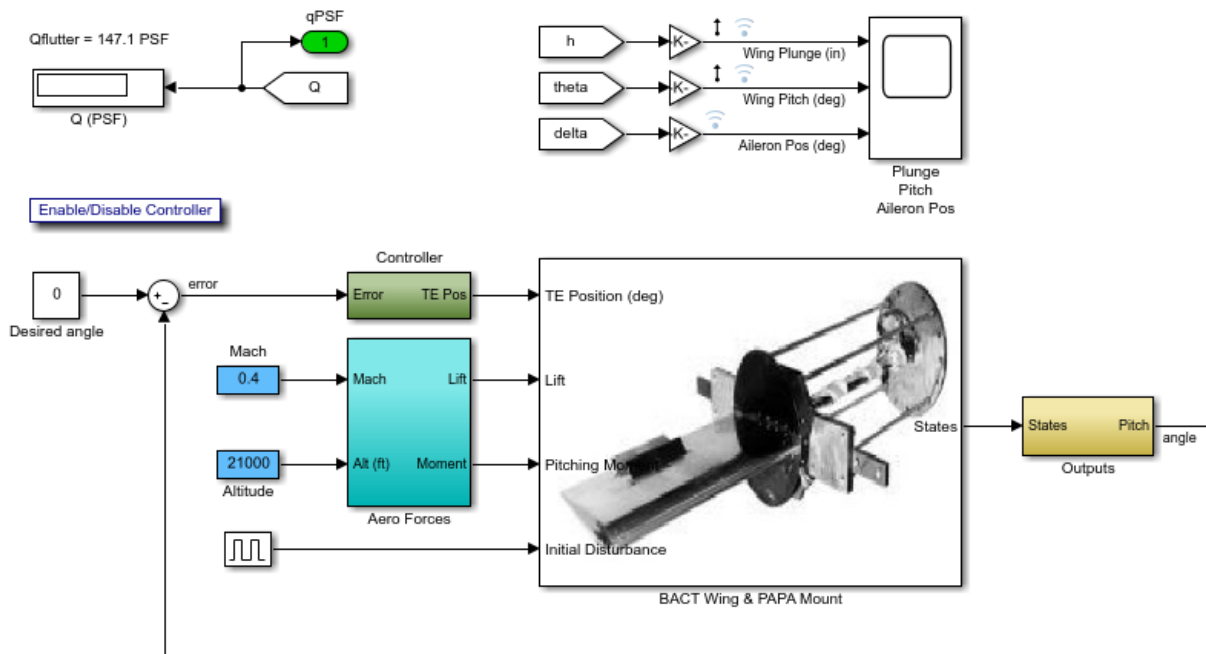
The test iterates over 16 combinations of Mach and Altitude. The test case uses custom criteria with Curve Fitting Toolbox™ to find the peaks of the wing pitch, and determine the damping ratio. If the damping ratio is not greater than zero, the assessment fails.

Running this test case requires

- Simulink® Test™
- Simscape Multibody™
- Aerospace Blockset™
- Curve Fitting Toolbox™

Open the model and the test file.

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



```
open(fullfile(matlabroot, 'examples', 'simulinktest', ...
    'sltestFlutterCriteriaTest.mldatx'))
```

### Custom Criteria Script

The test case custom criteria uses this script to verify that the damping ratio is greater than zero.

```
% Get time and data for pitch
```

```
Time = test.sltest_simout.get('sigOut').get('pitch').Values.Time(1:15000);
Data = test.sltest_simout.get('sigOut').get('pitch').Values.Data(1:15000);
```

```
% Find peaks
```

```
[~, peakIds] = findpeaks(Data, 'minpeakheight', 0.002, 'minpeakdistance', 50);
peakTime= Time(peakIds);
peakPos = Data(peakIds);
rn = peakPos(1)./peakPos(2:end);
L = 1:length(rn);
```

```
% Do curve fitting
fittedModel = exponentialFitAndPlot(L, rn);
delta = fittedModel.d;

% Find damping ratio
dRatio = delta/sqrt((2*pi)^2+delta^2);

% Make sure damping ratio is greater than 0
test.verifyGreaterThan(dRatio,0,'Damping ratio must be greater than 0');
```

### Test Results

Running the test case returns two conditions in which the damping ratio is greater than zero.

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

```
results =
```

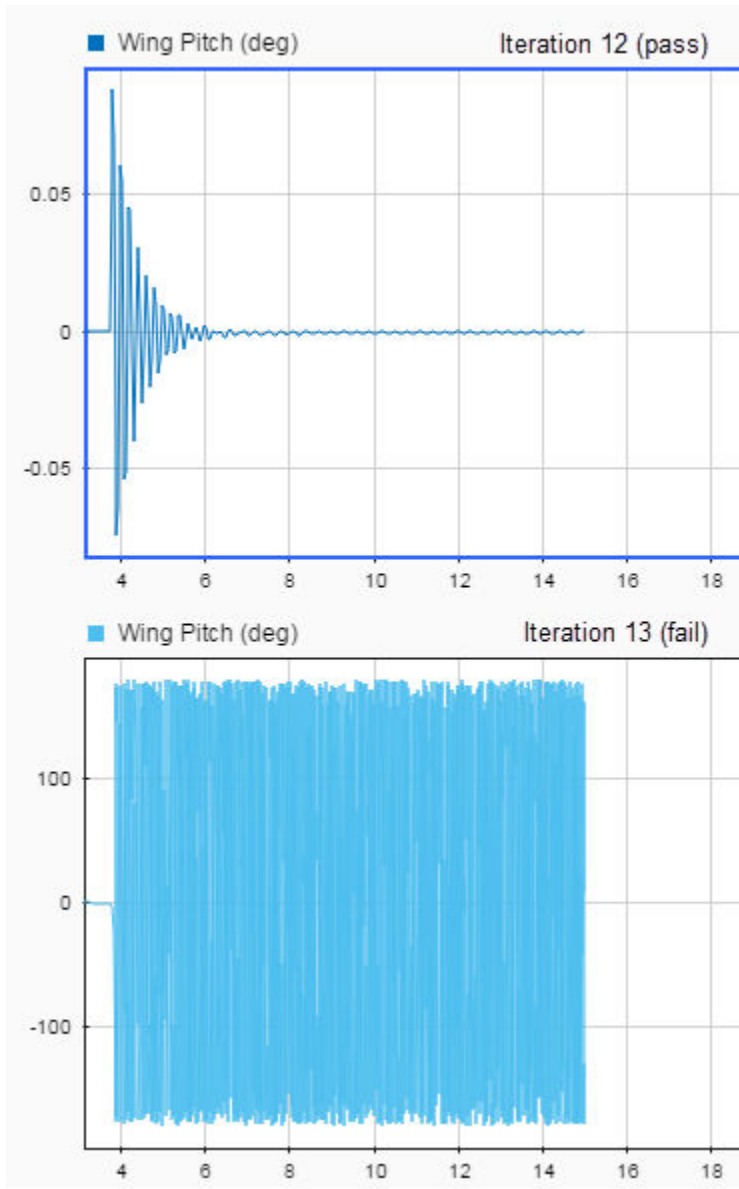
```
    ResultSet with properties:
```

```
        Name: 'Results: 2018-Aug-27 20:31:02'
        NumPassed: 14
        NumFailed: 2
        NumDisabled: 0
        NumIncomplete: 0
        NumTotal: 16
        NumTestCaseResults: 0
        NumTestSuiteResults: 0
        NumTestFileResults: 1
        Outcome: Failed
        StartTime: '2018-Aug-27 20:31:02'
        StopTime: '2018-Aug-27 20:35:01'
        Duration: 239
        CoverageResults: []
        Release: ''
```

▼ [I] Iteration13	✖
▶ [W] Sim Output (FlutterSuppressionSystem : r	
▼ [C] Custom Criteria Result	✖
<input checked="" type="checkbox"/> Damping ratio must be greater than 0	✖
▼ [I] Iteration14	✖
▶ [W] Sim Output (FlutterSuppressionSystem : r	
▼ [C] Custom Criteria Result	✖
<input checked="" type="checkbox"/> Damping ratio must be greater than 0	✖
▶ [I] Iteration15	✔

The wing pitch plots from iteration 12 and 13 show the difference between a positive damping ratio (iteration 12) and a negative damping ratio (iteration 13).





```
sltest.testmanager.close
```

```
close_system('sltestFlutterSuppressionSystemExample.slx',0)
```

## Custom Criteria Programmatic Interface Example

This example shows how to set and get custom criteria using the programmatic interface.

Before running this example, temporarily disable warnings that result from verification failures.

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

### Load a Test File and Get Test Case Object

```
tf = sltest.testmanager.load('AutopilotTestFile.mldatx');  
ts = getTestSuiteByName(tf, 'Basic Design Test Cases');  
tc = getTestCaseByName(ts, 'Requirement 1.3 Test');
```

### Create the Custom Criteria Object and Set Criteria

Create the custom criteria object.

```
tcCriteria = getCustomCriteria(tc)  
  
tcCriteria =  
    CustomCriteria with properties:  
  
    Enabled: 0  
    Callback: '% Return value: customCriteria...'
```

Create the custom criteria expression. This script gets the last value of the signal Phi and verifies that it equals 0.

```
criteria = ...  
    sprintf(['lastPhi = test.SimOut.get(''Signals_Req1_3'')', ...  
            '.get(''Phi'').Values.Data(end);\n', ...  
            'test.verifyEqual(lastPhi,0,[''Final: ',num2str(lastPhi),'.']);'])  
  
criteria =  
    'lastPhi = test.SimOut.get(''Signals_Req1_3'').get(''Phi'').Values.Data(end);  
    test.verifyEqual(lastPhi,0,[''Final: ',num2str(lastPhi),'.']);'
```

Set and enable the criteria.

```
tcCriteria.Callback = criteria;  
tcCriteria.Enabled = true;
```

### Run the Test Case and Get the Results

Run the test case.

```
tcResultSet = run(tc);
```

Get the test case results.

```
tcResult = getTestCaseResults(tcResultSet);
```

Get the custom criteria result.

```
ccResult = getCustomCriteriaResult(tcResult)
```

```
ccResult =  
    CustomCriteriaResult with properties:  
        Outcome: Failed  
        DiagnosticRecord: [1x1 sltest.testmanager.DiagnosticRecord]
```

Restore warnings from verification failures.

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

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

## See Also

### Related Examples

- “Test Models Using MATLAB Unit Test” on page 6-125
- “Create, Store, and Open MATLAB Figures” on page 6-122

## Create, Store, and Open MATLAB Figures

### In this section...

“Create a Custom Figure for a Test Case” on page 6-122

“Include Figures in a Report” on page 6-124

You can create figures using MATLAB commands to include with test results and reports. Enter the commands in a test case section that accepts MATLAB code. These sections include the test case **Custom Criteria** section, and callbacks that can execute with your test case.

If you include code that creates figures with your test case, you can:

- Display the figures after the test runs
- Store the figures with your test case
- Include them in a report
- Access stored figures from your test results

To specify this behavior, use the **Test File Options** section under the **Test File** settings.

- Select **Close all open figures at the end of execution** if you do not need to see the figures right after the test executes, for example, if you are storing the figures or including them in a report. Clear this check box if you are not storing the figures and you want to view them after the test executes.
- Select **Store MATLAB figures** if you want to save the figures with the test results. This option also enables you to open the figures from the results and to include them in a report.

After you run the test, the figures appear under **MATLAB Figures** in the test case results.

### Create a Custom Figure for a Test Case

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

- 1 Open the model `sldemo_absbrake`.
- 2 In the Test Manager, create a test file and name it `custom_figures`.

- 3 In the default test case, under **System Under Test**, set the model to `sldemo_absbrake`.
- 4 Under **Custom Criteria**, select the **function customCriteria(test)** check box and paste this code in the text box.

```

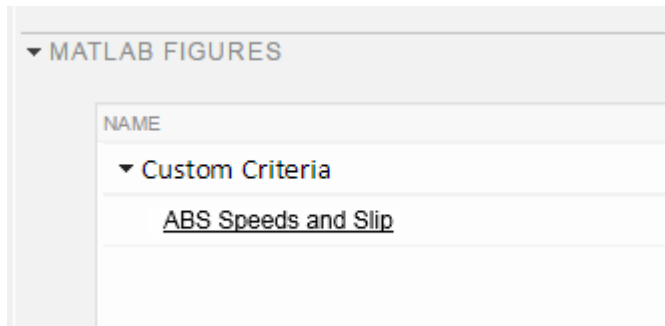
h = findobj(0,'Name','ABS Speeds and Slip');
if isempty(h)
    h=figure('Position',[26 100 452 700],...
            'Name','ABS Speeds and Slip',...
            'NumberTitle','off');
end
figure(h)
set(h,'DefaultAxesFontSize',8)

% Log data in sldemo_absbrake_output
out = test.sltest_simout.get('sldemo_absbrake_output');

% Plot wheel speed and car speed
subplot(3,1,1);
plot(out.get('yout').Values.Vs.Time, ...
      out.get('yout').Values.Vs.Data);
grid on;
title('Vehicle speed'); ylabel('Speed(rad/sec)'); xlabel('Time(sec)');
subplot(3,1,2);
plot(out.get('yout').Values.Ww.Time, ...
      out.get('yout').Values.Ww.Data);
grid on;
title('Wheel speed'); ylabel('Speed(rad/sec)'); xlabel('Time(sec)');
subplot(3,1,3);
plot(out.get('slp').Values.Time, ...
      out.get('slp').Values.Data);
grid on;
title('Slip'); xlabel('Time(sec)'); ylabel('Normalized Relative Slip');

```

- 5 Set the figure options for the test file `custom_figures`. Under **Test File Options**:
  - Select **Close all open figures at the end of execution**. This option closes figures created by your Test Manager MATLAB code.
  - Select **Store MATLAB figures**.
- 6 With the test case or the test file selected, click **Run**.
- 7 In the **Results and Artifacts** pane, select the test case under the results for this test run. Click the links under **MATLAB Figures** to see the plots generated when the test ran. The plot generated by the code you entered appears under **Custom Criteria**.



### Include Figures in a Report

You can select the **MATLAB Figures** option in the Create Test Results Report dialog box to include custom figures in your report. Alternatively, you can set report options under **Test File Options**. The **Test File Options** settings are saved with the test file.

- 1 Select the test file `custom_figures`.
- 2 Under **Test File Options**, select **Generate report after execution**. The section expands, displaying the same report options you can set using the dialog box.
- 3 To see the figures regardless of how the tests performed, set **Results for** to **All Tests**.
- 4 Select the **MATLAB figures** check box.
- 5 With the test file selected, run the test. Running the test generates the report and opens it in the PDF viewer.
- 6 Examine the report. The plot generated by the code you entered under **Custom Criteria** appears in the report section **Custom Criteria Plots**.

### See Also

`sltest.testmanager.Options` | `sltest.testmanager.TestCase.getOptions` | `sltest.testmanager.TestFile.getOptions` | `sltest.testmanager.TestSuite.getOptions`

### Related Examples

- “Export Test Results and Generate Reports” on page 7-9

# Test Models Using MATLAB Unit Test

## In this section...

“Overall Workflow” on page 6-125

“Considerations” on page 6-125

“Comparison of Test Nomenclature” on page 6-126

“Basic Workflow Using MATLAB® Unit Test” on page 6-127

You can use the MATLAB Unit Test framework to run tests authored in Simulink Test. Using the MATLAB Unit Test framework:

- Allows you to execute model tests together with MATLAB Unit Test scripts, functions, and classes.
- Enables model and code testing using the same framework.
- Enables integration with continuous integration (CI) systems, such as Jenkins™.

## Overall Workflow

To run tests with MATLAB Unit Test:

- 1** Create a `TestSuite` from the Simulink Test file.
- 2** Create a `TestRunner`.
- 3** Create plugin objects to customize the `TestRunner`. For example:
  - The `matlab.unittest.plugins.TAPPlugin` produces a results stream according to the Test Anything Protocol for use with certain CI systems.
  - The `sltest.plugins.ModelCoveragePlugin` specifies model coverage collection and makes coverage results accessible from the command line.
- 4** Add the plugins to the `TestRunner`.
- 5** Run the test using the `run` method, or run tests in parallel using the `runInParallel` method.

## Considerations

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

- If you disable a test in the Test Manager, the test is filtered using MATLAB Unit Test, and the result reflects a failed assumption.

## Comparison of Test Nomenclature

MATLAB Unit Test has analogous properties to the functionality in Simulink Test. For example,

- If the test case contains iterations, the MATLAB Unit Test contains parameterizations.
- If the test file or test suite contains callbacks, the MATLAB Unit Test contains one or more callbacks fixtures.

### Test Case Iterations and MATLAB Unit Test parameterizations

parameterization details correspond to properties of the iteration.

Simulink Test	MATLAB Unit Test
Iteration type: Scripted	parameterization property: ScriptedIteration
Iteration type: Table	parameterization property: TableIteration
Iteration name	parameterization Name
Test case iteration object	parameterization Value

### Test Callbacks and MATLAB Unit Test Fixtures

Fixtures depend on callbacks contained in the test file. Fixtures do not include test case callbacks, which are executed with the test case itself.

Callbacks in Simulink Test	Fixtures in MATLAB Unit Test
Test file callbacks	FileCallbacksFixture
Test suite callbacks	SuiteCallbacksFixture
File and suite callbacks	Heterogeneous CallbacksFixture, containing FileCallbacksFixture and SuiteCallbacksFixture
No callbacks	No fixture



## Basic Workflow Using MATLAB® Unit Test

This example shows how to create and run a basic MATLAB® Unit Test for a test file created in Simulink® Test™. You create a test suite, run the test, and display the diagnostic report.

Before running this example, temporarily disable warnings that result from verification failures.

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

1. Author a test file in the Test Manager, or start with a preexisting test file. For this example, `AutopilotTestFile` tests a component of an autopilot system against several requirements, using `verify` statements.

2. Create a `TestSuite` from the test file.

```
apsuite = testsuite('AutopilotTestFile.mldatx');
```

3. Run the test, creating a `TestResult` object. The command window returns warnings from the `verify` statement failures.

```
apresults = run(apsuite);
```

```
Setting up FileCallbacksFixture
Done setting up FileCallbacksFixture: Invoked setup callback of "AutopilotTestFile".
```

```
Running AutopilotTestFile > Basic Design Test Cases
```

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

```
-----
Framework Diagnostic:
-----
```

```
Failed criteria: Verification
```

```
--> Simulink Test Manager Results:
```

```
Results: 2018-Aug-27 20:30:09/AutopilotTestFile/Basic Design Test Cases/Re
```

```
=====
Done AutopilotTestFile > Basic Design Test Cases
-----
```

```
Tearing down FileCallbacksFixture  
Done tearing down FileCallbacksFixture: Invoked cleanup callback of "AutopilotTestFile"
```

Failure Summary:

Name	Failed	Incomplete
AutopilotTestFile > Basic Design Test Cases/Requirement 1.3 Test	X	

4. To view the details of the test, display the `Report` property of the `DiagnosticRecord` object. The record shows that a verification failed during the test.

```
apresults.Details.DiagnosticRecord.Report
```

```
ans =  
'Verification failed in AutopilotTestFile > Basic Design Test Cases/Requirement 1.3  
  
-----  
Framework Diagnostic:  
-----  
Failed criteria: Verification  
--> Simulink Test Manager Results:  
Results: 2018-Aug-27 20:30:09/AutopilotTestFile/Basic Design Test Case
```

Enable warnings.

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

## See Also

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

## Related Examples

- “Tests for Continuous Integration” on page 6-129
- “Run Tests for Various Workflows” (MATLAB)

## Tests for Continuous Integration

### In this section...

“Test a Model for Continuous Integration Systems” on page 6-129

“Model Coverage Results for Continuous Integration” on page 6-132

You can create model tests that are compatible with continuous integration (CI) systems such as Jenkins. To create CI-compatible results, run your Simulink Test files using MATLAB Unit Test.

To run CI-compatible tests, follow this general procedure:

- 1** Create a test suite from the MLDATX test file.
- 2** Create a test runner.
- 3** Create plugins for the test output or coverage results.
  - For test outputs, use the `TAPPlugin` or `XMLPlugin`.
  - For model coverage, use the `ModelCoveragePlugin` and `CoberturaFormat`. When collecting model coverage in Cobertura format:
    - Only top model coverage is reflected in the Cobertura XML.
    - Only model Decision coverage is reflected, and it is mapped to Condition elements in Cobertura XML.
- 4** Create plugins for CI-compatible output.
- 5** Add the plugins to the test output or coverage results.
- 6** Add the test output plugins or coverage result plugins to the test runner.
- 7** Run the test.

### Test a Model for Continuous Integration Systems

This example shows how to test a model, publish Test Manager results, and output results in TAP format with a single execution.

You use MATLAB® Unit Test to create a test suite and a test runner, and customize the runner with these plugins:

- `TestReportPlugin` produces a MATLAB Test Report.
- `TestManagerResultsPlugin` adds Test Manager results to the MATLAB Test Report.

- TAPPlugin outputs results to a TAP file.

The test case creates a square wave input to a controller subsystem and sweeps through 25 iterations of parameters **a** and **b**. The test compares the **alpha** output to a baseline with a tolerance of **0.0046**. The test fails on those iterations in which the output exceeds this tolerance.

Before running this example, ensure that the working folder is writable.

1. Open the Simulink® Test™ test file.

```
testfile = fullfile('f14ParameterSweepTest.mldatx');
sltest.testmanager.view;
sltest.testmanager.load(testfile);
```

2. In the Test Manager, configure the test file for reporting.

Under **Test File Options**, select **Generate report after execution**. The section expands, displaying several report options. For more information, see “Save Reporting Options with a Test File” on page 7-10.

3. Create a test suite from the Simulink® Test™ test file.

```
import matlab.unittest.TestSuite

suite = testsuite('f14ParameterSweepTest.mldatx');
```

4. Create a test runner.

```
import matlab.unittest.TestRunner

f14runner = TestRunner.withNoPlugins;
```

5. Add the TestReportPlugin to the test runner.

The plugin produces a MATLAB Test Report `F14Report.pdf`.

```
import matlab.unittest.plugins.TestReportPlugin

pdfFile = 'F14Report.pdf';
trp = TestReportPlugin.producingPDF(pdfFile);
addPlugin(f14runner,trp)
```

6. Add the TestManagerResultsPlugin to the test runner.

The plugin adds Test Manager results to the MATLAB Test Report.

```
import sltest.plugins.TestManagerResultsPlugin

tmr = TestManagerResultsPlugin;
addPlugin(f14runner,tmr)
```

7. Add the TAPPlugin to the test runner.

The plugin outputs to the F14output.tap file.

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

tapFile = 'F14output.tap';
tap = TAPPlugin.producingVersion13(ToFile(tapFile));
addPlugin(f14runner,tap)
```

8. Run the test.

Several iterations fail, in which the signal-baseline difference exceeds the tolerance criteria.

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

```
Generating report. Please wait.
```

```
    Preparing content for the report.
```

```
        Adding content to the report.
```

```
        Writing report to file.
```

```
Report has been saved to: C:\TEMP\Bdoc18b_943130_7372\ib632619\21\tp091df530\simulink
```

A single execution of the test runner produces two reports:

- A MATLAB Test Report that contains Test Manager results.
- A TAP format file that you can use with CI systems.

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

## Model Coverage Results for Continuous Integration

This example shows how to generate model coverage results for use with continuous integration. Coverage is reported in the Cobertura format. You run a Simulink® Test™ test file using MATLAB® Unit Test.

1. Import classes and create a test suite from the test file `AutopilotTestFile.mldatx`.

```
import matlab.unittest.TestRunner

aptest = sltest.testmanager.TestFile(fullfile(matlabroot, 'toolbox', 'simulinktest', ...
    'simulinktestdemos', 'AutopilotTestFile.mldatx'));
apsuite = testsuite(aptest.FilePath);
```

2. Create a test runner.

```
trun = TestRunner.withNoPlugins;
```

3. Set the coverage metrics to collect. This example uses decision coverage. In the Cobertura output, decision coverage is listed as condition elements.

```
import sltest.plugins.coverage.CoverageMetrics

cmet = CoverageMetrics('Decision', true);
```

4. Set the coverage report properties. This example produces a file `R13Coverage.xml` in the current working folder. Ensure your working folder has write permissions.

```
import sltest.plugins.coverage.ModelCoverageReport
import matlab.unittest.plugins.codecoverage.CoberturaFormat

rptfile = 'R13Coverage.xml';
rpt = CoberturaFormat(rptfile)

rpt =
    CoberturaFormat with no properties.
```

5. Create a model coverage plugin. The plugin collects the coverage metrics and produces the Cobertura format report.

```
import sltest.plugins.ModelCoveragePlugin

mcp = ModelCoveragePlugin('Collecting', cmet, 'Producing', rpt)
```

```
mcp =  
  ModelCoveragePlugin with properties:  
  
    RecordModelReferenceCoverage: '<default>'  
    MetricsSettings: [1x1 sltest.plugins.coverage.CoverageMetrics]
```

6. Add the coverage plugin to the test runner.

```
addPlugin(trun,mcp)
```

7. Run the test.

```
APResult = run(trun,apsuite)
```

```
APResult =  
  TestResult with properties:  
  
    Name: 'AutopilotTestFile > Basic Design Test Cases/Requirement 1.3 Test'  
    Passed: 0  
    Failed: 1  
    Incomplete: 0  
    Duration: 5.9468  
    Details: [1x1 struct]
```

```
Totals:  
  0 Passed, 1 Failed, 0 Incomplete.  
  5.9468 seconds testing time.
```

## See Also

TestRunner | TestSuite | matlab.unittest.plugins.TAPPlugin |  
matlab.unittest.plugins.TestReportPlugin |  
sltest.plugins.ModelCoveragePlugin |  
sltest.plugins.TestManagerResultsPlugin

## More About

- “Test Models Using MATLAB Unit Test” on page 6-125

## Filter Test Execution and Results

### In this section...

“Add Tags” on page 6-134

“Filter Tests and Results” on page 6-134

“Run Filtered Tests” on page 6-134

You can run a subset of tests or view a subset of test results by filtering test tags. Tags are a property of the test case, test suite, or test file.

### Add Tags

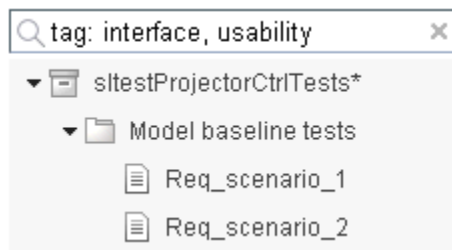
Add comma-separated tags to the **Tags** section in the Test Browser. Tags cannot contain spaces; spaces are corrected to commas.

▼ TAGS

safety, interface

### Filter Tests and Results

In the text box at the top of the **Test Browser** or **Results and Artifacts** pane, filter tests by entering `tag: id1, id2, ...` where `id1` and `id2` are example test tags. Enter multiple tags separated by commas to return tests containing any tag in the list.



### Run Filtered Tests

To run a subset of tests



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



# Test Manager Results and Reports

---

- “View Test Case Results” on page 7-2
- “Export Test Results and Generate Reports” on page 7-9
- “Customize Test Reports” on page 7-15
- “Append Code to a Test Report” on page 7-21
- “Results Sections” on page 7-24

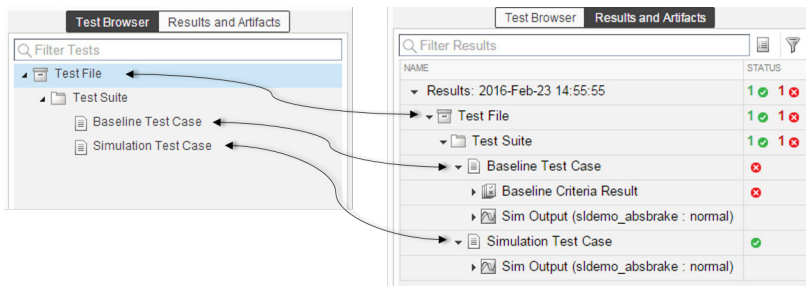
## View Test Case Results

### In this section...

“View Results Summary” on page 7-2

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

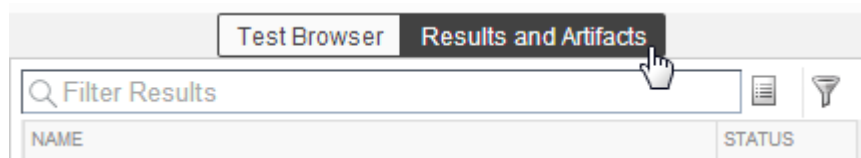
After a test case has finished running in the Test Manager, the test case result becomes available in the **Results and Artifacts** pane. Test results are organized in the same hierarchy as the test file, test suite, and test cases that were run from the **Test Browser** pane. In addition, the **Results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



## View Results Summary

The test case results tab gives a high-level summary and other information about an individual test case result. To open the test case results tab:

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



- 2 Double-click a test case result.

NAME	STATUS
▼ Results: 2016-Feb-23 14:55:55	1 ✓ 1 ✗
▼ Test File	1 ✓ 1 ✗
▼ Test Suite	1 ✓ 1 ✗
▼ Baseline Test Case	✗
▶ Baseline Criteria Result	✗

A tab opens containing the test case results information.

Baseline Test Case	
Name	<a href="#">Baseline Test Case</a>
Outcome	1
Start Time	02/23/2016 14:55:55
End Time	02/23/2016 14:55:56
Type	Baseline Test
Test File Location	C:\MATLAB\Test File.mldatx
Test Case Definition	
Rerun Test Case	
Baseline File	C:\MATLAB\test_capture.mat
Cause of Failure	<a href="#">Criteria evaluation resulted in failure.</a>
▶ Simulation Metadata	

▼ TEST REQUIREMENTS

▶ ITERATION SETTINGS

▼ ERRORS

▼ LOGS












▼ DESCRIPTION

## Visualize Test Case Simulation Output and Criteria

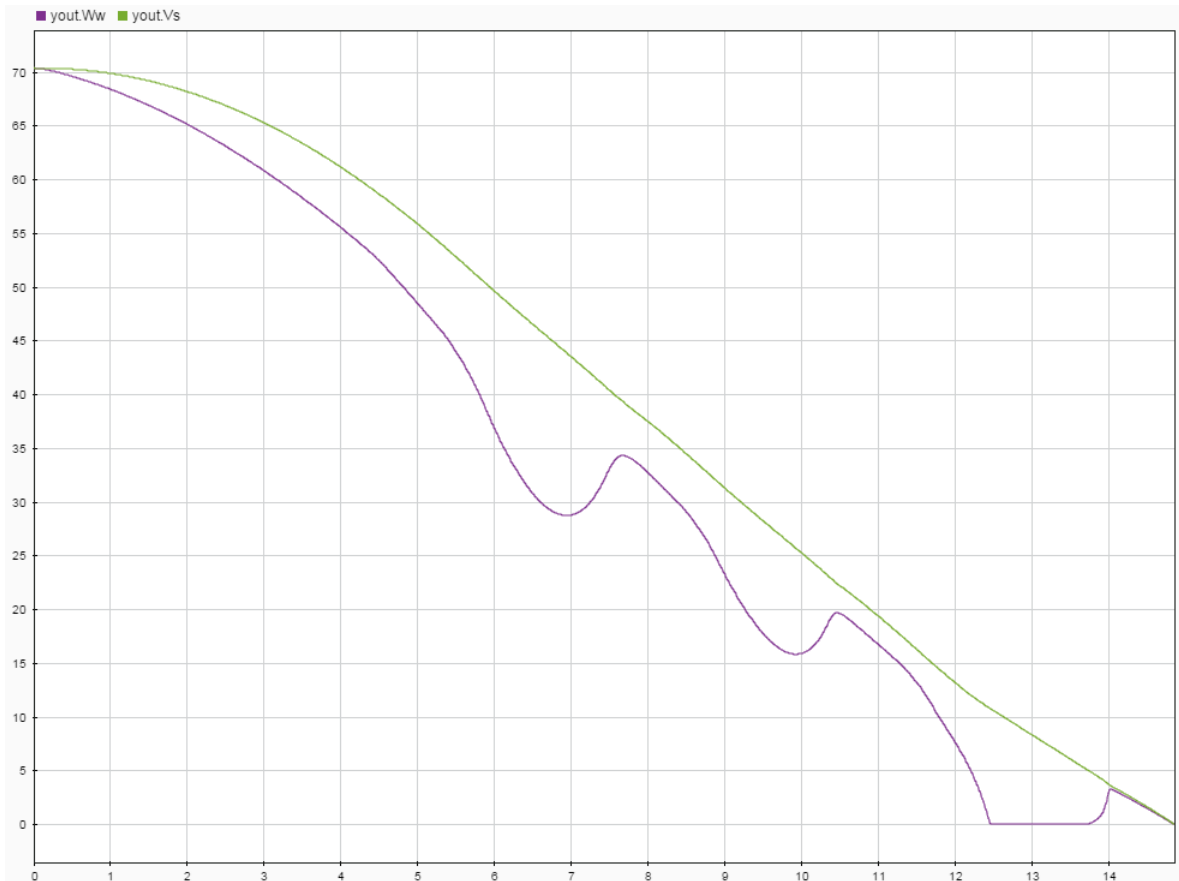
You can view signal data from simulation output or comparisons of signal data used in baseline or equivalence criteria.

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 : normal)		
<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.



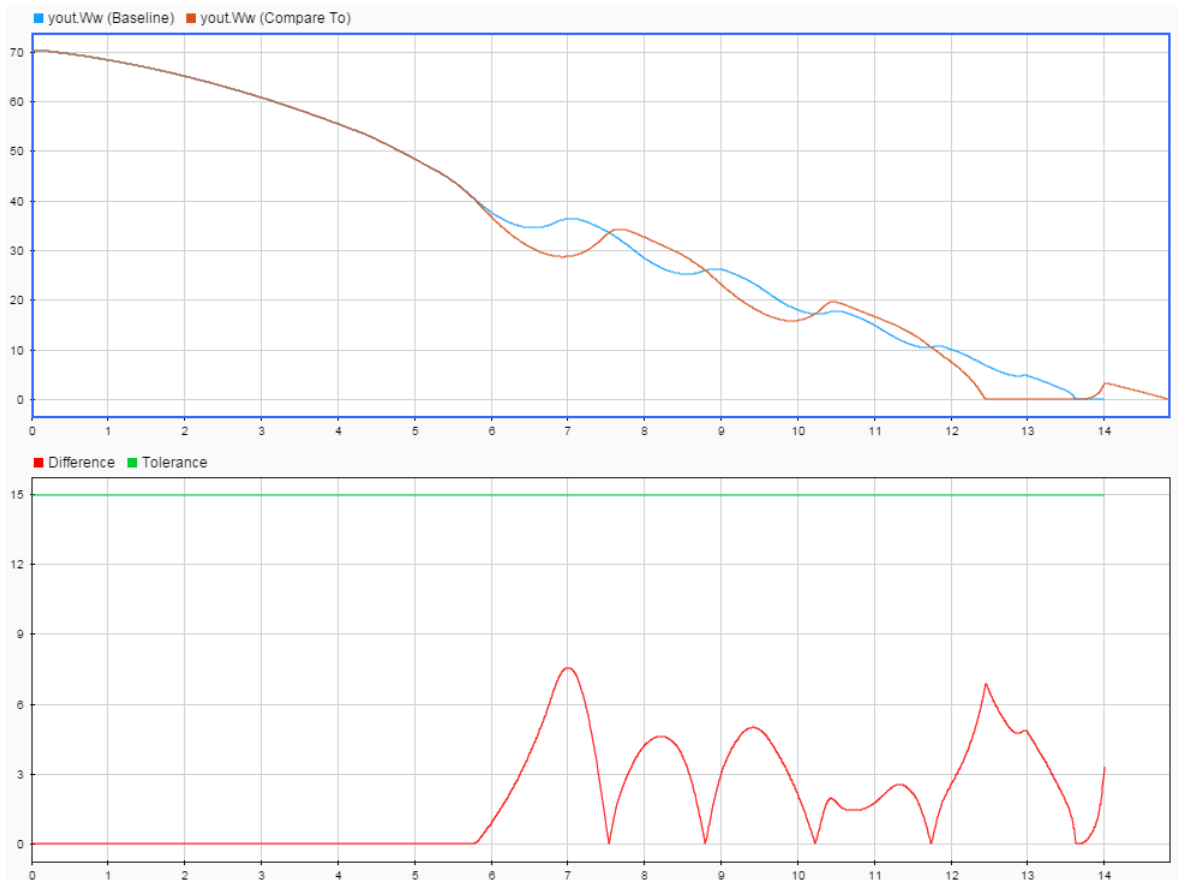
To view equivalence or baseline criteria comparisons:

- 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 : normal)		
<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
- “Save Reporting Options with a Test File” on page 7-10
- “Generate Reports Using Templates” on page 7-11
- “Generating a Test Results Report” on page 7-13

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

### Export Results

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

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

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

- Select criteria comparisons or simulation output to export signal data to the base workspace or to a MAT-file.

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

### Create a Test Results Report

Result reports contain report overview information, the test environment, results summaries with test outcomes, comparison criteria plots, and simulation output plots. You can customize the information included in the report, and you can save the report in three different file formats: ZIP (HTML), DOCX, and PDF.

- 1 In the **Results and Artifacts** pane, select results for a test file, test suite, or test case.

---

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

---

- 2 From the toolstrip, click **Report**.
- 3 Select the options to specify report contents.
- 4 Set **File Format** to the output format you want.
- 5 Click **Create**.

### Save Reporting Options with a Test File

You can generate a report every time you run a test case in a test file, using the same report settings each time. To generate a report each time you run the test, set options under **Test File Options**. These settings are saved with the test file.

- 1 In the **Test Browser** pane, select the test file whose report options you want to set.
- 2 Under **Test File Options**, select **Generate report after execution**. The section expands, displaying the same report options you can set using the dialog box.

- 3 Set the options. To include figures generated by callbacks or custom criteria, select **MATLAB figures**. For more information, see “Create, Store, and Open MATLAB Figures” on page 6-122.
- 4 Store the settings with your test file. Save the test file.
- 5 If you want to generate a report using these settings, select the test file and run the test.

## Generate Reports Using Templates

### Microsoft Word Format

If you have a MATLAB Report Generator™ license, you can create reports from a Microsoft Word template. The report can be a Microsoft Word or PDF document.

The report generator in Simulink Test fills information into rich text content controls in your Microsoft Word template document. For more information on how to use rich text content controls or customize part templates, see the MATLAB Report Generator documentation, such as “Add Holes in a Microsoft Word Template” (MATLAB Report Generator).

For a sample template, go to the path:


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

In the examples folder, open the file `Template.dotx`.

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

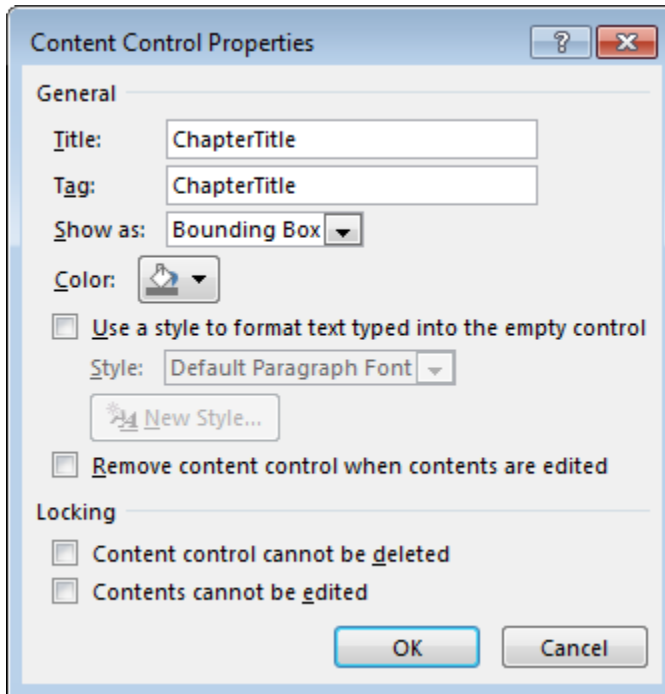
- `ChapterTitle` — report title
- `ChapterTestPlatform` — version of MATLAB used to execute tests
- `ChapterTOC` — test results table of contents
- `ChapterBody` — test results

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



« ChapterTitle Click here to enter text. ChapterTitle »

To change the control name, right-click the rich text content control and select **Properties**. Specify the control name, `ChapterTitle` or 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 toolstrip, click **Report**.
- 4 Select the report options.
- 5 Select DOCX or PDF for the **File Format**.
- 6 Specify the full path and file name of your Microsoft Word template.
- 7 Click **Create**.

## PDF or HTML Formats

If you have a MATLAB Report Generator license, you can create reports from a PDF or HTML template, using a PDFTX or HTMTX file. To generate a report from the Test Manager using a PDF or HTML template:

- 1 In the Test Manager, select the **Results and Artifacts** pane.
- 2 Select results for a test file, test suite, or test case in the **Results and Artifacts** pane.
- 3 From the toolstrip, click **Report**.
- 4 Select the report options.
- 5 Select ZIP or PDF for the **File Format**. Selecting ZIP generates an HTML report.
- 6 Specify the full path and file name of your template. For PDF, use a PDFTX file. For HTML, use an HTMTX file. For more information on creating templates, see “Templates” (MATLAB Report Generator).
- 7 Click **Create**.

## Generating a Test Results Report

Report test results for a baseline test. Copyright 2015 The MathWorks, Inc.

This example shows how to generate a test results report from the test manager using a baseline test case. The model used for this example is `sltestTestManagerReportsExample`. Switch to a directory with write permissions.

### Load and run the test file

Load and run the test file programmatically using the test manager. The test file contains a baseline test case that fails when it is run. The baseline criteria specified in the baseline test case does not match the model simulation, which makes the test case fail.

```
exampleFile = fullfile(matlabroot, ...  
    'toolbox', 'simulinktest', 'simulinktestdemos', ...  
    'sltestTestManagerReportsTestSuite.mldatx');  
sltest.testmanager.load(exampleFile);  
baselineObj = sltest.testmanager.run;
```

### Generate the report

Generate a report of the test case results using the results set object. The report is saved as a ZIP and will show all test results. The report opens when it is completed.

```
sltest.testmanager.report(baselineObj, 'baselineReport.zip', 'IncludeTestResults', 0, 'In
```

View the report when it is finished generating. Notice that the overall baseline test case fails. The signals in baseline criteria do not match, which causes the test failure. You can view the signal comparison plots in the report to verify the failure.

```
sltest.testmanager.clear;  
sltest.testmanager.clearResults;
```

## See Also

### Related Examples

- “Templates” (MATLAB Report Generator)
- “Create, Store, and Open MATLAB Figures” on page 6-122



# Customize Test Reports

## In this section...

“Inherit the Report Class” on page 7-15

“Method Hierarchy” on page 7-15

“Modify the Class” on page 7-17

“Generate a Report Using the Custom Class” on page 7-19

You can choose how to format and aggregate test results by customizing reports. Use the `sltest.testmanager.TestResultReport` class to create a subclass and then use the properties and methods to customize how the Test Manager generates the results report. You can change font styles, add plots, organize results into tables, include model images, and more. Using the custom class, requires a MATLAB Report Generator license.

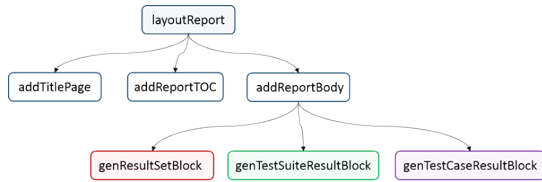
## Inherit the Report Class

To customize the generated report, you must inherit from the `sltest.testmanager.TestResultReport` class. After you inherit from the class, you can modify the properties and methods. To inherit the class, add the class definition section to a new or existing MATLAB script. The subclass is your custom class name, and the superclass that you inherit from is `sltest.testmanager.TestResultReport`. For more information about creating subclasses, see “Design Subclass Constructors” (MATLAB). Then, add code to the inherited class or methods to create your customizations.

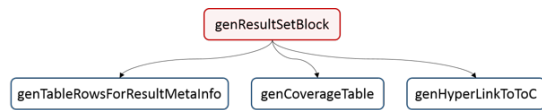
```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    %
    % Report customization code here
    %
end
```

## Method Hierarchy

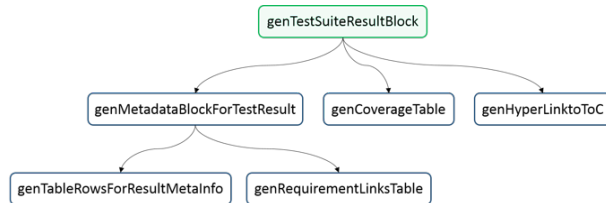
When you create the subclass, the derived class inherits methods from the `sltest.testmanager.TestResultReport` class. The body of the report is separated into three main groups: the result set block, the test suite result block, and the test case result block.



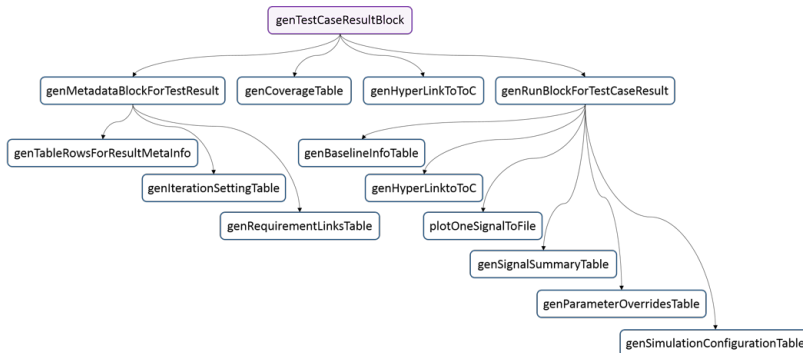
The result set block contains the result set table, the coverage table, and links to the table of contents.



The test suite result block contains the test suite results table, the coverage table, requirements links, and links to the table of contents.



The test case result block contains the test case and test iterations results table, the coverage table, requirements links, signal output plots, comparison plots, test case settings, and links to the table of contents.



## Modify the Class

To insert your own report content or change the layout of the generated report, modify the inherited class methods. For general information about modifying methods, see “Modify Inherited Methods” (MATLAB).

A simple modification to the generated report could be to add some text to the title page. The method used here is `addTitlePage`.

```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    methods
        function this = CustomReport(resultObjects, reportFilePath)
            this@sltest.testmanager.TestResultReport(resultObjects, ...
                reportFilePath);
        end
    end

    methods(Access=protected)
        function addTitlePage(obj)
            import mlreportgen.dom.*;

            % Add a custom message
            label = Text('Some custom content can be added here');
            append(obj.TitlePart,label);

            % Call the superclass method to get the default behavior
            addTitlePage@sltest.testmanager.TestResultReport(obj);
        end
    end
end
```

[Click here for a code file of this example.](#)

A more complex modification of the generated report is to include a snapshot of the model that was tested.

```
% class definition
classdef CustomReport < sltest.testmanager.TestResultReport
    methods
        function this = CustomReport(resultObjects,reportFilePath)
            this@sltest.testmanager.TestResultReport(resultObjects,reportFilePath);
        end
    end
end
```

```
methods(Access=protected)
% Method to customize test case/iteration result section in the report
function docPart = genTestCaseResultBlock(obj,result)
    % result: A structure containing test case or iteration result
    import mlreportgen.dom.*;

    % Call the superclass method to get the default behavior
    docPart = genTestCaseResultBlock@sltest.testmanager.TestResultReport(...
                                                obj,result);

    % Get the test case result data for putting in the report
    tcrObj = result.Data;

    % Insert model screenshot at the test case result level
    if isa(tcrObj, 'sltest.testmanager.TestCaseResult')

        % Initialize model name
        modelName = '';

        % Check in the test case result if it has model information. If
        % not, it means there were iterations in the test case or a
        % model was not used.
        testSimMetaData = tcrObj.SimulationMetaData;

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

        % Get iteration results
        iterResults = getIterationResults(tcrObj);

        % Get the model name in case test case had iterations
        if (~isempty(iterResults))
            modelName = iterResults(1).SimulationMetaData.modelName;
        end

        % Insert model snapshot. This will not work for harnesses. With
        % minimal changes we can also open the harness used for
        % testing.
        if (~isempty(modelName))
            try
                open_system(modelName);
                snapObj = SLPrint.Snapshot;
```

```

snapObj.Target = modelName;
snapObj.Format = 'png';
snapObj.FileName = fullfile(tempdir,modelName);
if exist(snapObj.FileName, 'file')
    delete( snapObj.FileName );
end
snapObj.snap;
outputFileName = snapObj.FileName;
outputFileName = [outputFileName '.png'];
para = sltest.testmanager.ReportUtility.genImageParagraph(...
    outputFileName,...
    '5.2in', '3.7in');
append(docPart,para);
catch
end
end
end
end
end
end
end

```

[Click here for a code file of this example.](#)

## Generate a Report Using the Custom Class

After you customize the class and methods, use the `sltest.testmanager.report` to generate the report. You must use the 'CustomReportClass' name-value pair for the custom class, specified as a string. For example:

```

% Generate the result set from imported data
result = sltest.testmanager.importResults('demoResults.mldatx');

% Specify the report file name and path
filePath = 'testreport.zip';

% Generate the report using the custom class
sltest.testmanager.report(result,filePath, ...
    'Author','MathWorks',...
    'Title','Test',...
    'IncludeMLVersion',true,...
    'IncludeTestResults',int32(0),...
    'CustomReportClass','CustomReport',...
    'LaunchReport', true);

```

Alternatively, you can create your custom report using the Test Manager report dialog box. Select a test result, click the **Report** button on the toolbar, and specify the custom report class in the Create Test Result Report dialog box. For the Test Manager to use the custom report class, the class must be on the MATLAB path.

### See Also

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

### Related Examples

- “Design Subclass Constructors” (MATLAB)

## Append Code to a Test Report

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

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

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

### Navigate to the Example Folder

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

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

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

### Report Customization Class

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

```
open(className)
```

### Load the Test Results and Create the Test Report

1. Load the test results file.

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

2. Create the test report using the customization.

```
sltest.testmanager.report(result, filePath, 'CustomReportClass', className, ...
    'IncludeTestResults', 0)
```

3. The report appends the S-Function wrapper code:

## S-Function Wrapper

```
/*
 * Include Files
 *
 */
#ifdef(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

/* %%%-SFUNWIZ_wrapper_includes_Changes_BEGIN --- EDIT HERE TO _END */
#include <math.h>
#include "RejectDoublePress.h"
/* %%%-SFUNWIZ_wrapper_includes_Changes_END --- EDIT HERE TO _BEGIN */
#define u_width 1
#define y_width 1
/*
 * Create external references here.
 *
 */
/* %%%-SFUNWIZ_wrapper_externs_Changes_BEGIN --- EDIT HERE TO _END */
/* extern double func(double a); */
/* %%%-SFUNWIZ_wrapper_externs_Changes_END --- EDIT HERE TO _BEGIN */

/*
 * Output functions
 *
 */
void RejectDoublePress_sfun_Outputs_wrapper(const boolean_T *AccelResSwIn,
                                             const boolean_T *CoastSetSwIn,
                                             boolean_T *AccelResSwOut,
                                             boolean_T *CoastSetSwOut)
```



For more information on report customization, see [Customize Generated Reports](#)

```
sltest.testmanager.clearResults;  
sltest.testmanager.close;
```

## Results Sections

In this section...
"Summary" on page 7-25
"Test Requirements" on page 7-25
"Iteration Settings" on page 7-26
"Errors" on page 7-26
"Logs" on page 7-26
"Description" on page 7-26
"Parameter Overrides" on page 7-26
"Coverage Results" on page 7-26

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

✕ **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 test requirements linked to the test case. See “Requirements” on page 6-97 for more information on linking requirements to test cases.

### Iteration Settings

If you are using iterations to run test cases, then this section appears in the results. For more information about test iterations, see “Run Combinations of Tests Using Iterations” on page 6-70.

### Errors

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

### Logs

This section displays simulation warnings captured from the Simulink Diagnostic Viewer.

### Description

You can include notes about the test results here. These notes are saved with the results.

### Parameter Overrides

A list of parameter overrides specified in the test case under **Parameter Overrides**. If parameter overrides are not specified, then this section is not shown in the results summary.

### Coverage Results

If you collect coverage in your test, then the coverage results appear in this section. Coverage results are aggregated at the test file, test suite, and test file level. For more information about coverage, see “Collect Coverage in Tests” on page 6-81.

# Real-Time Testing

---

- “Test Models in Real Time” on page 8-2
- “Reuse Desktop Test Cases for Real-Time Testing” on page 8-13

## 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

- Baseline or equivalence comparisons can fail because of missing data or time-shifted data from the real-time target computer. When investigating real-time test failures, look for time shifts or missing data points.
- You cannot override the real-time execution sample time for applications built from models containing a Test Sequence block. The code generated for the Test Sequence block contains a hard-coded sample time. Overriding the target computer sample time can produce unexpected results.
- Your target computer must have a file system to use `verify` statements and test case logging.

## Complete Basic Model Testing

Real-time testing often takes longer than comparative model testing, especially if you execute a suite of real-time tests that cover several scenarios. Before executing real-time tests, complete requirements-based testing using desktop simulation. Using the desktop simulation results:

- Debug your model or make design changes that meet requirements.
- Debug your test sequence. Use the debugging features in the test sequence editor. See “Debug a Test Sequence” on page 3-69.
- Update your requirements and add corresponding test cases.

## Set up the Target Computer

Real-time testing requires a standalone target computer. Simulink Test only supports target computers running Simulink Real-Time™. For more information, see:

- “Development Computer Setup and Configuration” (Simulink Real-Time)
- “Troubleshooting in Simulink Real-Time” (Simulink Real-Time)

## Configure the Model or Test Harness

Real-time applications require specific configuration parameters and signal properties.

### Code Generation

A real-time test case requires a real-time system target file. In the model or harness configuration parameters, in the **Code Generation** pane, set the **System target file** to `slrt.tlc` to generate system target code.

If your model requires a different system target file, you can set the parameter using a test case or test suite callback. After the real-time test executes, set the parameter to its original setting with a cleanup callback. For example, this callback opens the model and sets the system target file parameter to `slrt.tlc` for the model `sltestProjectorController`.

```
open_system(fullfile(matlabroot, 'toolbox', 'simulinktest', ...  
'simulinktestdemos', 'sltestProjectorController'));  
set_param('sltestProjectorController', 'SystemTargetFile', 'slrt.tlc');
```

### Data Import/Export Format

Models must use a data format other than `dataset`. To set the data format:

- 1 Open the configuration parameters.
- 2 Select the **Data Import/Export** pane.
- 3 Select the **Format**.

### Log Signals from Real-Time Execution

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

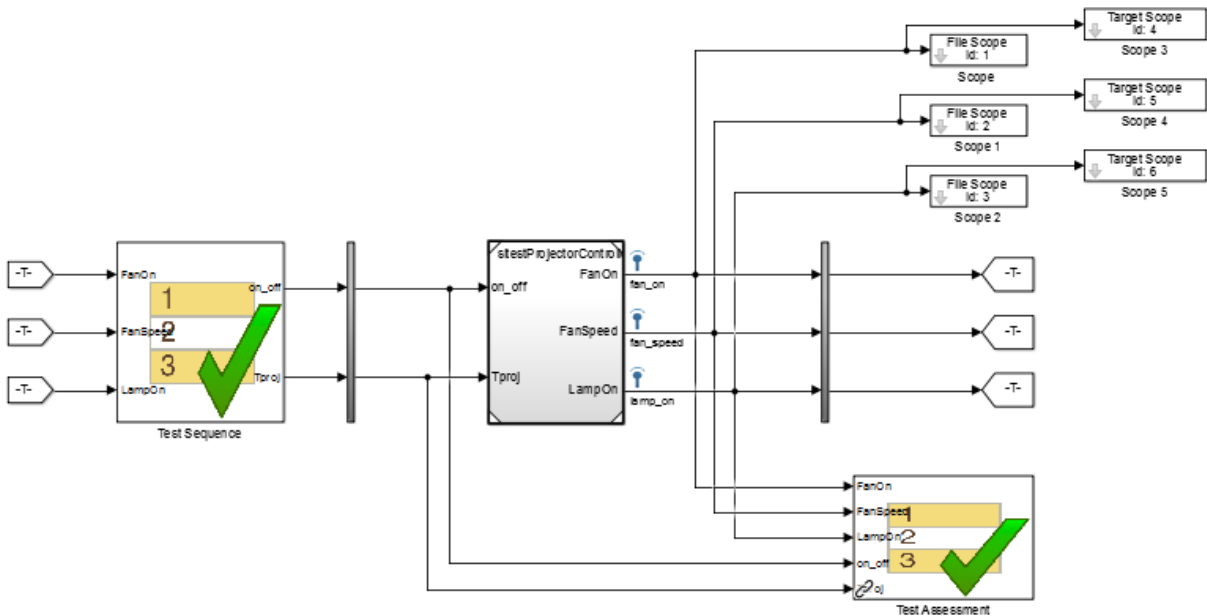
- Enable signal logging in the Configuration Parameters, in the Data Import/Export pane.
- Connect signals to Scope blocks from the Simulink Real-Time block library. Set the **Scope type** property to File.



- Name each signal of interest using the signal properties. Unnamed signals can be assigned a default name which does not match the name of the baseline or equivalence signal.

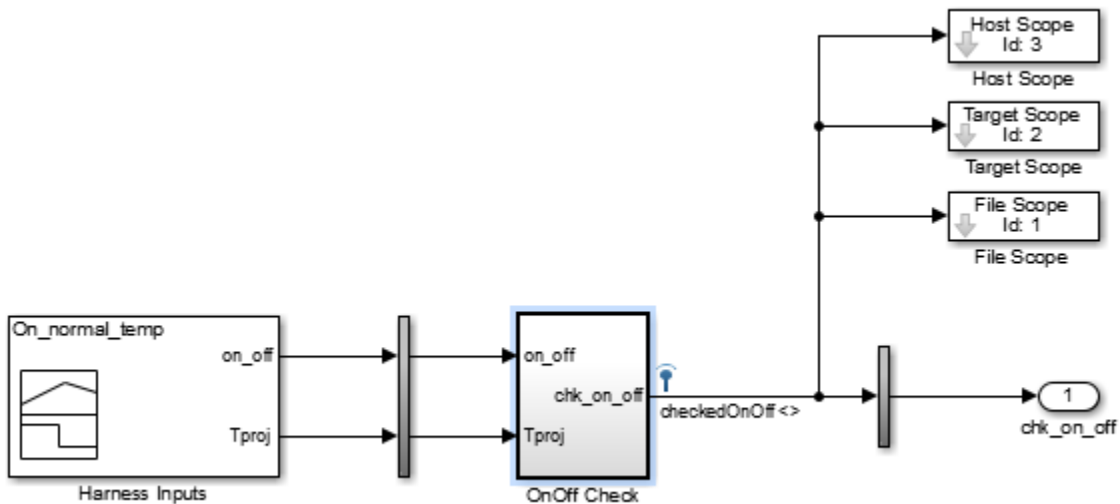
In this example test harness, the logged signals:

- Have explicit names.
- Use file scopes to return signal data to the Test Manager.
- Use target scopes to display data on the target computer.



## View Signals During Real-Time Execution

To display signals on the target computer during real-time execution, add target scopes to your test harness. To display signals in the Simulink Real-Time Explorer, add host scopes. This test harness includes both target and host scopes for signal visualization. See Scope.



## Add Test Cases for Real-Time Testing

Use the Test Manager to create real-time test cases. In the toolstrip, click **New > Real-Time Test**.

### Test Type

You can select a baseline, equivalence, or simulation real-time test. For simulation test types, verify statements serve as pass/fail criteria in the test results. For equivalence and baseline test types, the equivalence or baseline criteria also serve as pass/fail criteria.

- **Baseline** — Compares the signal data returned from the target computer to the baseline in the test case. To compare a real-time execution result to a model simulation result, add the model baseline result to the real-time test case and apply optional tolerances to the signals.
- **Equivalence** — Compares signal data from a simulation and a real-time test, or two real-time tests. To run a real-time test on the target computer, then compare results to a model simulation:
  - Select **Simulation 1 on target**.

- Clear **Simulation 2 on target**.

The test case displays two simulation sections, **Simulation 1** and **Simulation 2**.

Comparing two real-time tests is similar, except that you select both simulations on target. In the **Equivalence Criteria** section, you can capture logged signals from the simulation and apply tolerances for pass/fail analysis.

- **Simulation:** Assesses the test result using only `verify` statements and real-time execution. If no `verify` statements fail, and the real-time test executes, the test case passes.

### Load Application From

Using this option, specify how you want to load the real-time application. The real-time application is 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

Test Case Execution Step (first to last)	Load Application From		
	Model	Target Application	Target Computer
Executes post-load callback	Yes	No	No
Sets Signal Builder group	Yes	No	No
Builds 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

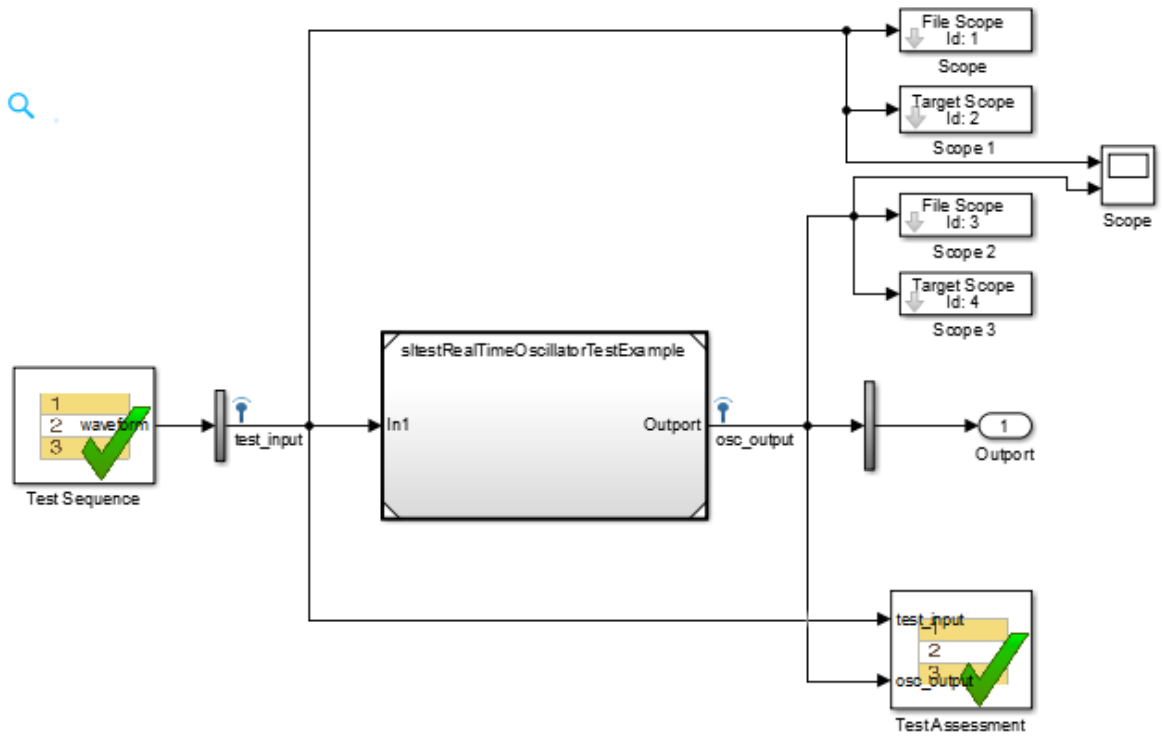
Real-time tests offer a **Pre-start real-time application** callback which executes commands just before the application executes on the target computer. Real-time test callbacks execute in a sequence along with the model load, build, download, and execute steps. Callbacks and step execution depends 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	<b>Preload callback</b>	<b>Preload callback</b>	<b>Preload callback</b>
	<b>Post-load callback</b>	—	—
	<b>Pre-start real-time callback</b>	<b>Pre-start real-time callback</b>	<b>Pre-start real-time callback</b>
Executes last	<b>Cleanup callback</b>	<b>Cleanup callback</b>	<b>Cleanup callback</b>

## 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 Combinations of Tests Using Iterations” on page 6-70. 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;	1. true	step_2 ▼
<b>step_2</b> waveform =square(et*frequency)*0.5 + 0.5;		

In the test file `SettlingTestCases`, the real-time test scripted iterations cover a frequency sweep from 5 Hz to 35 Hz. The script iterates the value of frequency in the Test Sequence block.

```
%% Iterate over frequencies to determine best oscillator settings

% Create parameter sets
freq = 5.0:1.0:35.0;

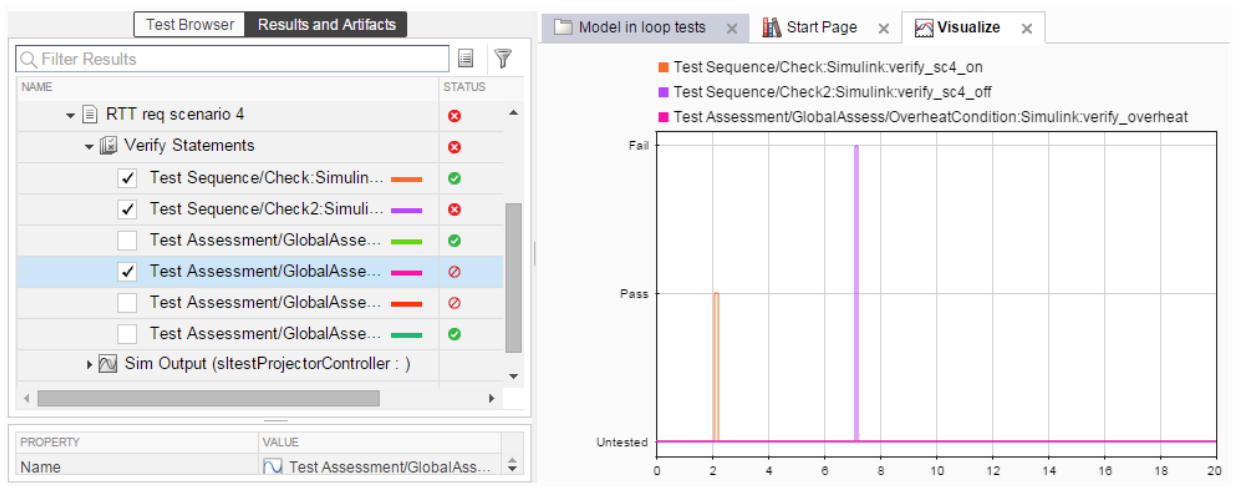
for i_iter = 1:length(freq)
    % Create iteration object
    testItr = sltestiteration();

    % Set parameters
    setVariable(testItr, 'Name', 'frequency', 'Source', 'Test Sequence', ...
        'Value', freq(i_iter));

    % Register iteration
    addIteration(sltest_testCase, testItr);
end
```

## Assess Real-Time Execution Using `verify` Statements

In addition to baseline and equivalence signal comparisons, you can assess real-time test execution using `verify` statements. A `verify` statement assesses a logical expression and returns results to the Test Manager. Use `verify` inside a Test Sequence or Test Assessment block. See “Run-Time Assessments” on page 3-49.



## See Also

### Related Examples

- "Test Real-Time Application" (Simulink Real-Time)



# Reuse Desktop Test Cases for Real-Time Testing

## Convert Desktop Test Cases to Real-Time

In the Test Manager, you can reuse test cases for real-time testing by converting desktop test cases to real-time test cases. For convenience, data can be stored externally so that each test case accesses common inputs and baseline data. The overall workflow is as follows:

- 1 Create a baseline, equivalence, or simulation test case with external inputs. For baseline tests, add baseline data from external files.
- 2 In the Test Manager, select the test case in the **Test Browser**.
- 3 Copy the test case. Right-click the test case and select **Copy**.
- 4 Paste the new test case into a test suite.
- 5 Rename the new test case.
- 6 Right-click the new test case, and select **Convert to > Real-Time Test**. For equivalence tests, select which simulation (simulation 1 or simulation 2) to run in real time.
- 7 Select the **Target Computer** and **Load Application From** options.
- 8 Ensure that the model settings are compatible with real-time test execution. For more information, see “Development Computer Setup and Configuration” (Simulink Real-Time).

## Use External Data for Real-Time Tests

You can simplify test input data management by defining the input data in an external MAT or Excel file. Map the data to root inports in your model or test harness for desktop simulation. When you convert the desktop simulation test case into a real-time test, the test case uses the same inport mapping.

Using external data depends on how your test case loads the real-time application:

### Load Real-Time Application from Model

If you are using external data for a real-time test, loading the real-time application from the model gives you the option of using an Excel file, MAT file, or CSV file. The external data is built into the application, and you can rerun the application from the target application or target computer.

In the **System Under Test** section, set the application to load from Model. In the **Inputs** section of the test case, click **Add**, and select an Excel file, MAT file, or CSV file. Map the

data to your model inputs. For more information on input mapping, see “Use External Inputs in Test Cases” on page 6-42.

### **Load Real-Time Application from Target Application or Target Computer**

After running the test from the model, you can run the test from the target application or target computer without recompiling. The application uses the input mapping from when the test ran from the model.

You can map external data to a test case loaded from the target application or target computer, without first running from the model. The external data must be in a MAT file, in the same format used if the test is loaded from the model. In the **System Under Test** section, select to load the application from the **Target Application or Target Computer**. In the **Inputs** section, click **Add** and select a MAT file. The Input string is not editable.

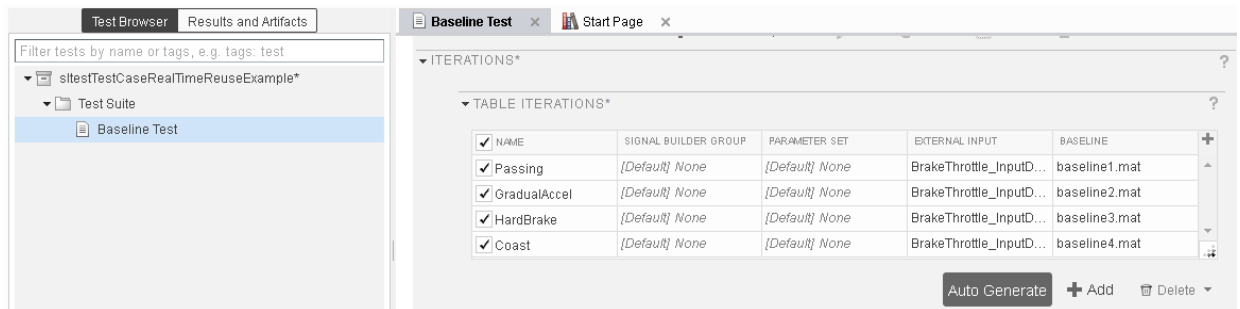
### **Example**

This example shows a basic desktop test case reuse workflow using external input data defined in an Excel file. You run the baseline test case on the desktop, update the baseline data, convert a copy of the test case to a real-time test, then run the test case on a target computer. This example runs only on Windows systems.

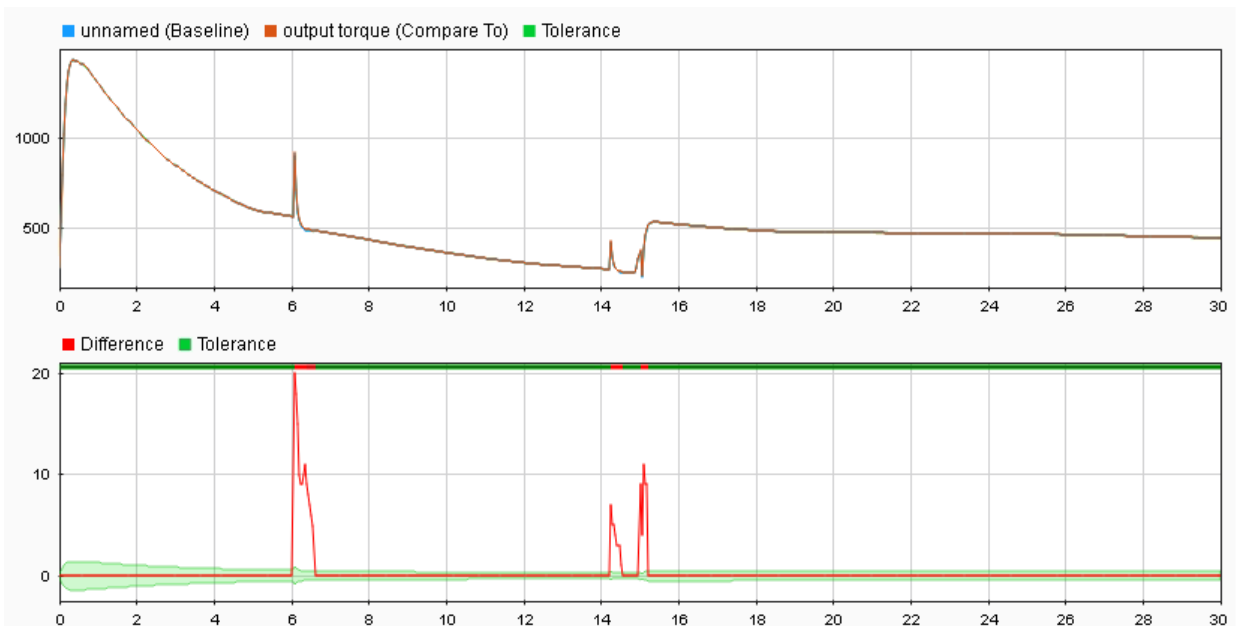
- 1 Open the test file.

```
tf = sltest.testmanager.TestFile(fullfile(matlabroot, 'examples', ...  
'simulinktest', 'sltestTestCaseRealTimeReuseExample.mldatx'));  
sltest.testmanager.load(tf.Name);  
sltest.testmanager.view;
```

The test file runs a transmission shift controller algorithm through four iterations, each corresponding to a different test scenario: passing, gradual acceleration, hard braking, and coasting. There is baseline data associated with each scenario for the signals `vehicle speed` and `output torque`.

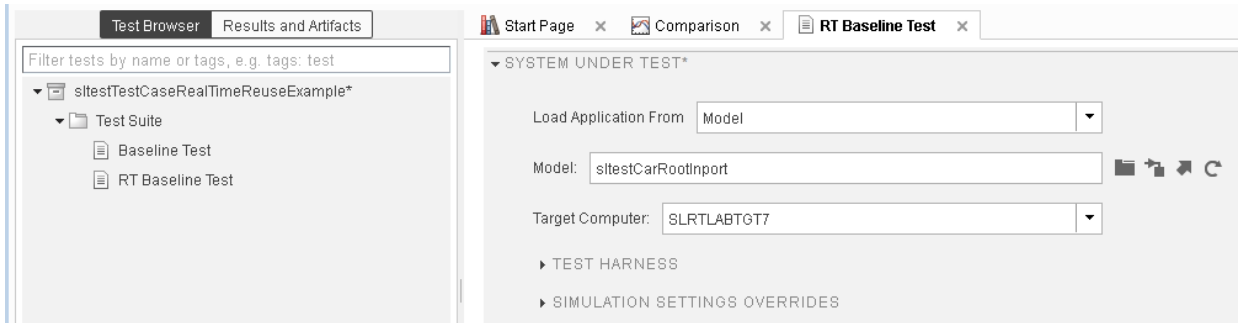


- 2 Run the baseline test.
- 3 Under the Baseline Criteria result, select **output torque** under the **Passing** result to view the comparison. The **Passing** result fails due to transient signals that fall outside the relative tolerance.

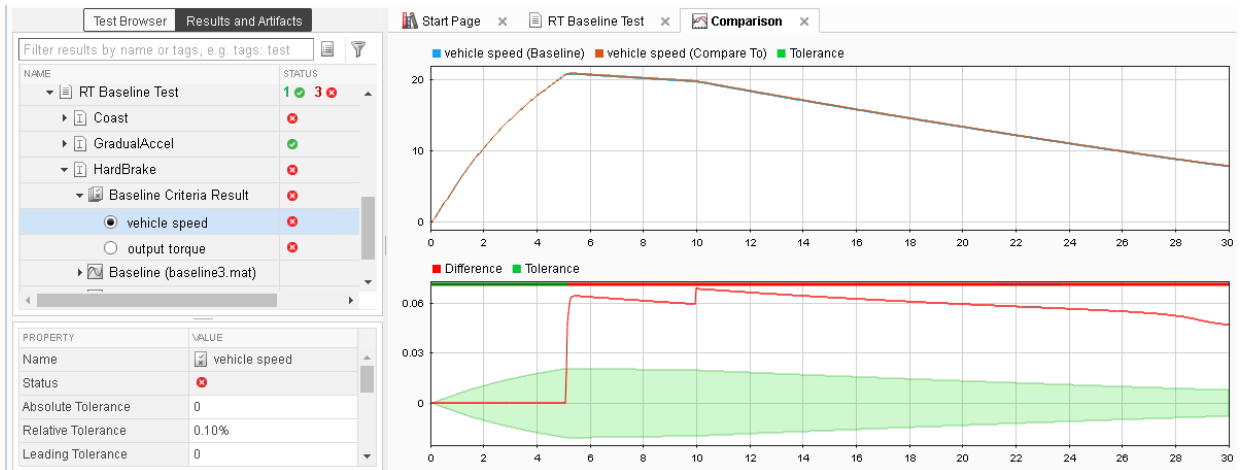


- 4 Assume that these transient signals are not significant, and update the baseline data:
  - 1 Click **Next Failure**. The first failure region is bounded by data cursors.
  - 2 Click **Update Baseline + Update selected signal region**, and confirm that you want to overwrite the data.

- 3 Repeat this process for the other two failure regions.
- 5 Copy and convert the baseline test case to a real-time test:
  - 1 In the Test Browser, right-click Baseline Test and select Copy.
  - 2 Paste the new test case under the test suite.
  - 3 Rename the new tests case RT Baseline Test.
  - 4 Right-click RT Baseline Test and select **Convert to > Real-Time Test**.
- 6 Run the real-time test case:
  - 1 Set the **Target Computer**.
  - 2 Set the system under test to load from Model.



- 3 Run the RT Baseline Test test case.
- 7 In this example, several of the scenarios fail due to timing impacts on the data output. For example, in the HardBrake iteration, the vehicle speed output falls outside the relative tolerance after the brake is applied. To resolve this failure, you could:
  - Increase the relative tolerance for the real-time test.
  - Create a separate set of baseline data for the real-time test.



## See Also

### Related Examples

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



# Verification and Validation

---

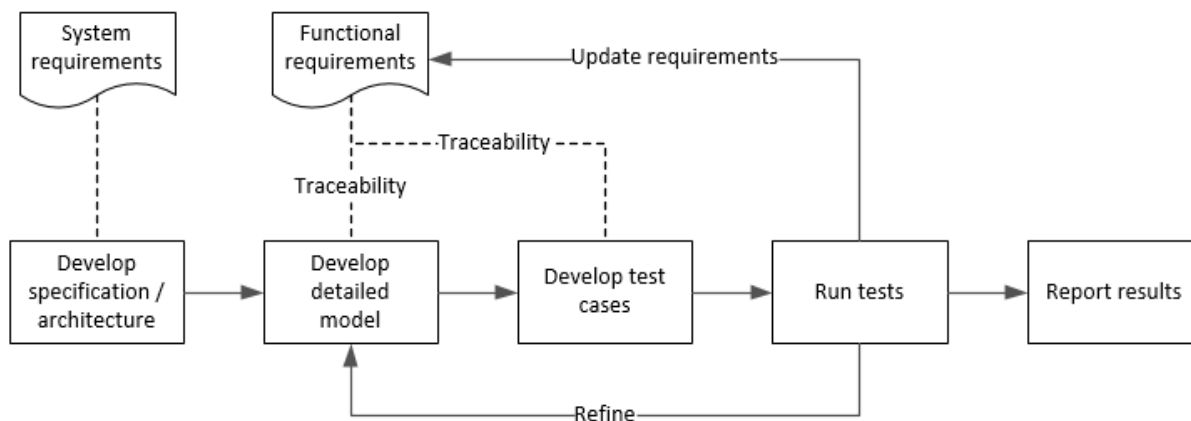
- “Test Model Against Requirements and Report Results” on page 9-2
- “Analyze a Model for Standards Compliance and Design Errors” on page 9-8
- “Perform Functional Testing and Analyze Test Coverage” on page 9-11
- “Analyze Code and Test Software-in-the-Loop” on page 9-15

## Test Model Against Requirements and Report Results

### Requirements - Test Traceability Overview

Traceability between requirements and test cases helps you interpret test results and see the extent to which your requirements are verified. You can link a requirement to elements that help verify it, such as test cases in the Test Manager, verify statements in a Test Sequence block, or Model Verification blocks in a model. When you run tests, a pass/fail summary appears in your requirements set.

This example demonstrates a common requirements-based testing workflow for a cruise control model. You start with a requirements set, a model, and a test case. You add traceability between the tests and the safety requirements. You run the test, summarize the verification status, and report the results.



In this example, you conduct a simple test of two requirements in the set:


- That the cruise control system transitions to disengaged from engaged when a braking event has occurred
- That the cruise control system transitions to disengaged from engaged when the current vehicle speed is outside the range of 20 mph to 90 mph.



## Display the Requirements and Test Case

- 1 Create a copy of the project in a working folder. The project contains data, documents, models, and tests. Enter:

```
path = fullfile(matlabroot,'toolbox','shared','examples',...  
'verification','src','cruise')  
run(fullfile(path,'slVerificationCruiseStart'))
```

- 2 In the project models folder, open the `simulinkCruiseAddReqExample.slx` model.
- 3 Display the requirements. Click the  icon in the lower-right corner of the model canvas, and select **Requirements**. The requirements appear below the model canvas.
- 4 Expand the requirements information to include verification and implementation status. Right-click a requirement and select **Verification Status** and **Implementation Status**.

The screenshot displays a Simulink model window titled 'simulinkCruiseAddReqExample'. The model contains a central block labeled 'Compute target speed' with two internal states. It has five inputs: 'CruiseOnOff' (boolean), 'Brake' (boolean), 'Speed' (single), 'CoastSetSw' (boolean), and 'AccelResSw' (boolean). It has two outputs: 'engaged' (boolean) and 'tspeed' (single). The outputs are connected to external blocks labeled '1' and '2'.

Below the Simulink window is the 'Requirements - simulinkCruiseAddReqExample' browser. It shows a tree view of requirements and a table with the following data:

Index	ID	Summary	Implemented	Verified
simulinkCruiseChartReqs				
1	Architecture	Architecture		
1.1	A 1.1	Enable Disable Switch		
1.2	A 1.2	Set Speed / Decelerate Button		
1.3	A 1.3	Resume Speed / Accelerate Butt...		
1.4	A 1.4	Engaged Output		
1.5	A 1.5	Target Speed Output		
1.6	A 1.6	Vehicle Speed Input		
1.7	A 1.7	Vehicle Brake Input		
2	Functional Requirements	Functional Requirements		
3	Safety Requirements	Safety Requirements		
3.1	S 3.1	Vehicle braking disengages syst...		
3.2	S 3.2	System engagement speed limit...		
3.3	S 3.3	Target speed limitations		
3.4	S 3.4	Speed outside limits disengages ...		

On the right side, the 'Requirement: A 1.2' details pane is visible, showing the title 'Set Speed/Decelerate Button' and a description: 'The controller shall have an input button to: set the target speed to the current vehicle speed when the cruise control is **not engaged (active)** decelerate (reduce) the target speed when the cruise control is **engaged (active)**'.

- 5 Open the Simulink Test file `slReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager.

### Link Requirements to Tests

Link the requirements to the test case.

- 1 In the Requirements Browser, select requirement S 3.1.
- 2 In the Test Manager, expand the test file and select the **Safety Tests** test case. Expand the **Requirements** section.
- 3 In the **Requirements** section, select **Add > Link to Selected Requirement**.

The requirements browser displays the verification-type link.

3	Safety Requirements	Safety Requirements
3.1	S 3.1	Vehicle braking disengages system
3.2	S 3.2	System engagement speed limitations
3.3	S 3.3	Target speed limitations
3.4	S 3.4	Speed outside limits disengages system

Verified by: [Safety Tests](#)

Comments

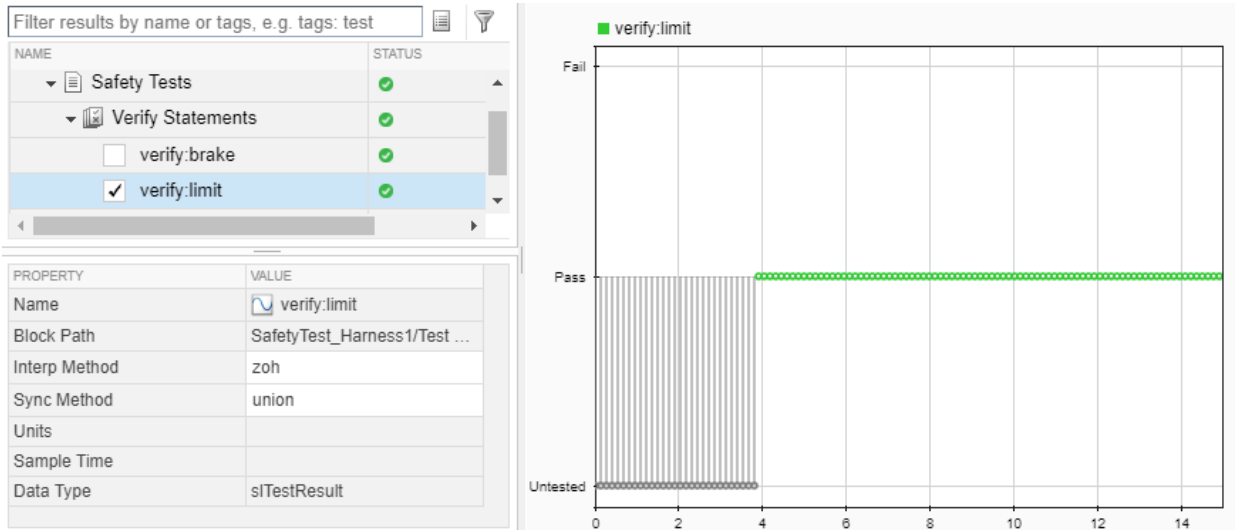
- 4 Also add a link for item S 3.4.

## Run the Test

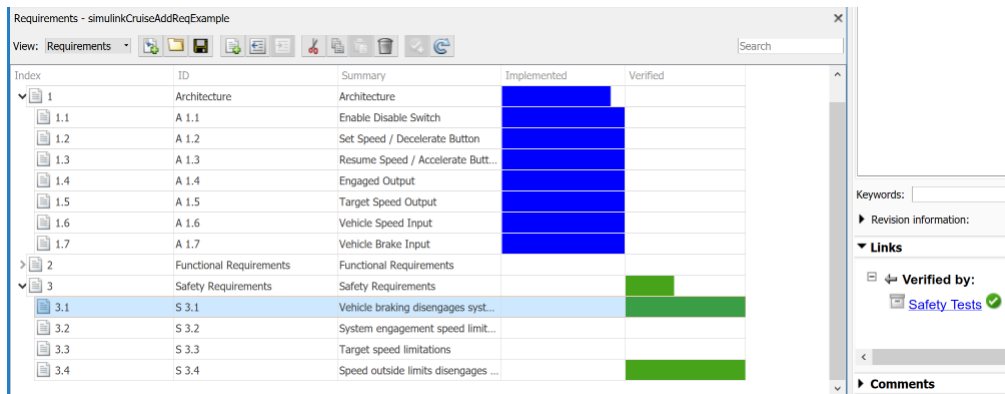
- 1 The test case uses a test harness `SafetyTest_Harness1`. In the test harness, a test sequence sets the input conditions and checks the model behavior:
  - The `BrakeTest` sequence engages the cruise control, then applies the brake. It includes the `verify` statement
 

```
verify(engaged == false,...
      'verify:brake',...
      'system must disengage when brake applied')
```
  - The `LimitTest` sequence engages the cruise control, then ramps up the vehicle speed until it exceeds the upper limit. It includes the `verify` statement.
 

```
verify(engaged == false,...
      'verify:limit',...
      'system must disengage when limit exceeded')
```
- 2 Run the test case. In the Test Manager toolstrip, click **Run**.
- 3 When the test finishes, expand the **Verify Statements** results. The Test Manager results show that both assessments pass, and the plot shows the detailed results of each `verify` statement.



- 4 In the Requirements Browser, right-click a requirement and select **Refresh Verification Status** to show the passing test results for each requirement.



## Report the Results

- 1 Create a report using a custom Microsoft Word template.
  - a From the Test Manager results, right-click the test case name. Select **Create Report**.

- b In the Create Test Result Report dialog box, set the options:
    - Title — SafetyTest
    - Results for — All Tests
    - File Format — DOCX
    - For the other options, keep the default selections.
  - c For the **Template File**, select the ReportTemplate.dotx file in the **documents** project folder.
  - d Enter a file name and select a location for the report.
  - e Click **Create**.
- 2 Review the report.
- a In the **Test Case Requirements** section, click the link to trace to the requirements document.
  - b The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.

Name	Data Type	Units	Sample Time	Interp	Sync	Link to Plot
✔ Test Sequence/.../Verify:verify(engaged == false)	siTestResult			zoh	union	<a href="#">Link</a>
✔ Test Sequence/.../VerifyHigh:verify(engaged == false)	siTestResult			zoh	union	<a href="#">Link</a>

## See Also

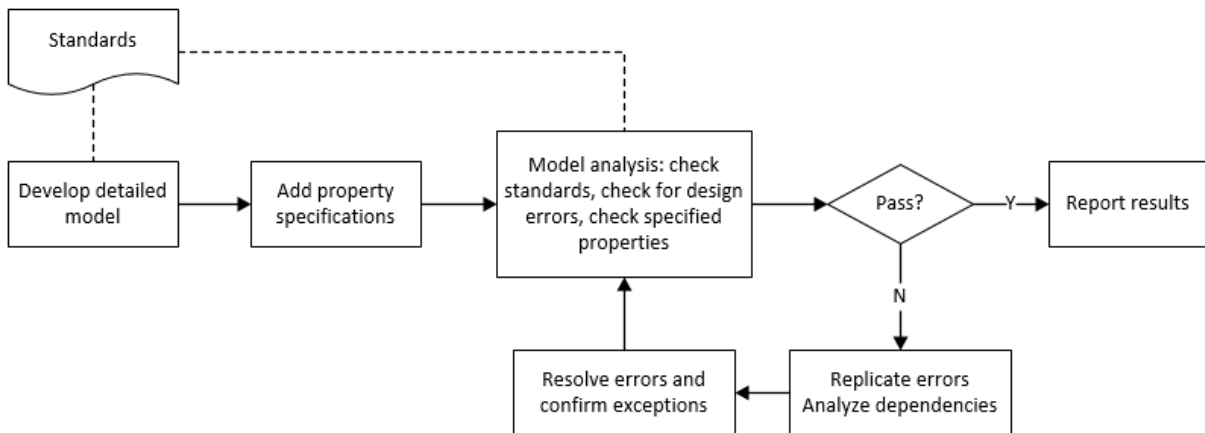
### Related Examples

- “Link Tests to Requirements” on page 1-2
- “Validate Requirements Links in a Model” (Simulink Requirements)
- “Customize Requirements Traceability Report for Model” (Simulink Requirements)

## Analyze a Model for Standards Compliance and Design Errors

### Standards and Analysis Overview

During model development, check and analyze your model to increase confidence in its quality. Check your model against standards such as MAAB style guidelines and high-integrity system design guidelines such as DO-178 and ISO 26262. Analyze your model for errors, dead logic, and conditions that violate required properties. Using the analysis results, update your model and document exceptions. Report the results using customizable templates.



### Check Model for Style Guideline Violations and Design Errors

This example shows how to use the Model Advisor to check a cruise control model for MathWorks® Automotive Advisory Board (MAAB) style guideline violations and design errors. Select checks and run the analysis on the model. Iteratively debug issues using the Model Advisor and rerun checks to verify that it is in compliance. After passing your selected checks, report results.

## Check Model for MAAB Style Guideline Violations

In Model Advisor, you can check that your model complies with MAAB modeling guidelines.

- 1 Create a copy of the project in a working folder. On the command line, enter

```
path = fullfile(matlabroot,'toolbox','shared','examples',...  
'verification','src','cruise')  
run(fullfile(path,'slVerificationCruiseStart'))
```

- 2 Open the model. On the command line, enter

```
open_system simulinkCruiseErrorAndStandardsExample
```

- 3 In the model window, select **Analysis > Model Advisor > Model Advisor**.

- 4 Click OK to choose `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.

- 5 Check your model for MAAB style guideline violations using Simulink Check.

- a In the left pane, in the **By Product > Simulink Check > Modeling Standards > MathWorks Automotive Advisory Board Checks** folder, select:

- **Check for indexing in blocks**
- **Check for prohibited blocks in discrete controllers**
- **Check model diagnostic parameters**

- b Right-click the **MathWorks Automotive Advisory Board Checks** node, and then select **Run Selected Checks**.

- c Click **Check model diagnostic parameters** to review the configuration parameter settings that violate MAAB style guidelines.

- d In the right pane, click the parameter links to update the values in the Configuration Parameters dialog box.

- e To verify that your model passes, rerun the check. Repeat steps c and d, if necessary, to reach compliance.

- f To generate a results report of the Simulink Check checks, select the **MathWorks Automotive Advisory Board Checks** node, and then, in the right pane click **Generate Report...**

## Check Model for Design Errors

While in Model Advisor, you can also check your model for hidden design errors using Simulink Design Verifier.

- 1** In the left pane, in the **By Product > Simulink Design Verifier** folder, select **Design Error Detection**.
- 2** In the right pane, click **Run Selected Checks**.
- 3** After the analysis is complete, expand the **Design Error Detection** folder, then select checks to review warnings or errors.
- 4** In the right pane, click **Simulink Design Verifier Results Summary**. The dialog box provides tools to help you diagnose errors and warnings in your model.
  - a** Review the results on the model. Click **Highlight analysis results on model**. Click the **Compute target speed** subsystem, outlined in red. The Simulink Design Verifier Results Inspector window provides derived ranges that can help you understand the source of an error by identifying the possible signal values.
  - b** Review the harness model. The Simulink Design Verifier Results Inspector window displays information that an overflow error occurred. To see the test cases that demonstrate the errors, click **View test case**.
  - c** Review the analysis report. In the Simulink Design Verifier Results Inspector window, click **Back to summary**. To see a detailed analysis report, click **HTML** or **PDF**.

## See Also

### Related Examples

- “Check for Compliance Using the Model Advisor and Edit-Time Checking” (Simulink Check)
- “Collect Model Metrics Using the Model Advisor” (Simulink Check)
- “Run a Design Error Detection Analysis” (Simulink Design Verifier)
- “Prove Properties in a Model” (Simulink Design Verifier)

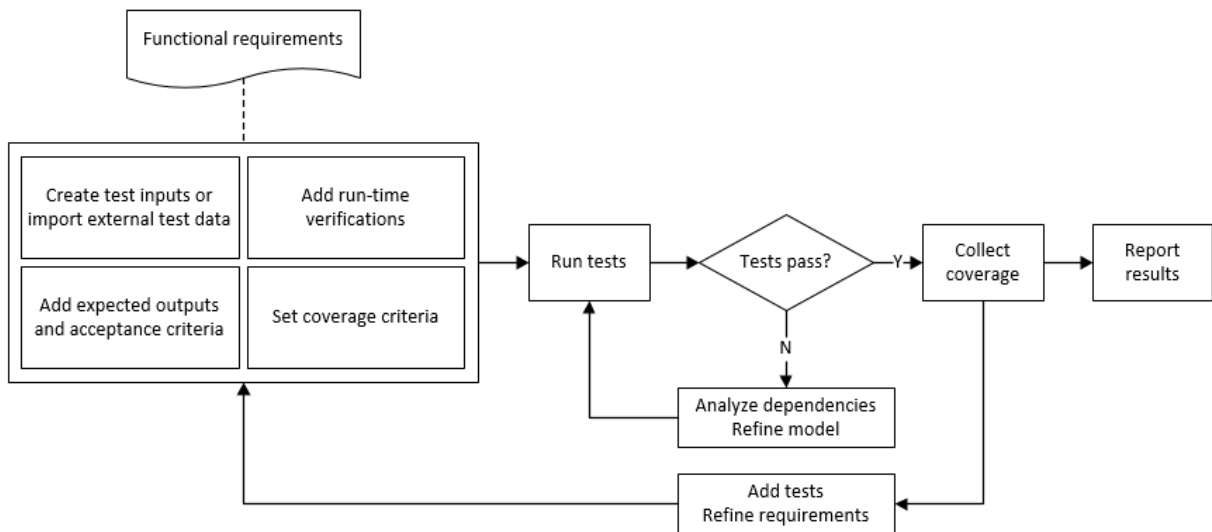


# Perform Functional Testing and Analyze Test Coverage

## Functional Testing and Coverage Analysis Overview

Functional testing starts with building test cases based on requirements. These tests can cover key aspects of your design and verify that individual model components meet requirements. Test cases include inputs, expected outputs, and acceptance criteria.

By collecting individual test cases within test suites, you can run functional tests systematically. To check for regression, add baseline criteria to the test cases and test the model regularly. Coverage measurement reflects the extent to which these tests have fully exercised the model. Coverage measurement also helps you to add tests and requirements to meet coverage targets.



## Incrementally Increase Test Coverage Using Test Case Generation

This example shows a functional testing-based testing workflow for a cruise control model. You start with a model that has tests linked to an external requirements document, analyze the model for coverage in Simulink Coverage, incrementally increase coverage with Simulink Design Verifier, and report the results.

### Explore the Test Harness and the Model

- 1 Create a copy of the project in a working folder. At the command line, enter:

```
path = fullfile(matlabroot, 'toolbox', 'shared', 'examples', ...  
'verification', 'src', 'cruise')  
run(fullfile(path, 'slVerificationCruiseStart'))
```

- 2 Open the model and the test harness. At the command line, enter:

```
open_system simulinkCruiseAddReqExample  
sltest.harness.open('simulinkCruiseAddReqExample', 'SafetyTest_Harness1')
```

- 3 Load the test suite from “Test Model Against Requirements and Report Results” on page 9-2. At the command line, enter:

```
sltest.testmanager.load('slReqTests.mldatx')  
sltest.testmanager.view
```

- 4 Open the test sequence block. The sequence tests:

- That the system disengages when the brake pedal is pressed
- That the system disengages when the speed exceeds a limit

Some test sequence steps are linked to a requirements document `simulinkCruiseChartReqs.docx`.

### Measure Model Coverage

- 1 In the test manager, enable coverage collection for the test case.
  - a Open the test manager. In the Simulink menu, click **Analysis > Test Manager**.
  - b In the **Test Browser**, click the `slReqTests` test file.
  - c Expand **Coverage Settings**.
  - d Under **COVERAGE TO COLLECT**, select **Record coverage for referenced models**.
  - e Under **COVERAGE METRICS**, select **Decision**, **Condition**, and **MCDC**.

▼ COVERAGE SETTINGS

▼ COVERAGE TO COLLECT

Record coverage for system under test

Record coverage for referenced models

COVERAGE METRICS

Decision  Condition

MCDC  Lookup Table






Signal Range  Signal Size



Simulink Design Verifier  Saturation on integer overflow

Relational Boundary

- 2 Run the test. On the test manager toolbar, click **Run**.
- 3 When the test finishes, in the Test Manager, navigate to the test case. The aggregated coverage results show that the example model achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.

▼ AGGREGATED COVERAGE RESULTS

ANALYZED MODEL	REPORT CO...	DECISION	CONDITION	MCDC
 <a href="#">simulinkCruiseAddReqExample</a>	 31	50% 	41% 	25% 

 Add Tests for Missing Coverage  Export

## Generate Tests to Increase Model Coverage

- 1 Use Simulink Design Verifier to generate additional tests to increase model coverage. Select the test case in the **Results and Artifacts** and open the aggregated coverage results section.
- 2 Select the test results from the previous section and then click **Add Tests for Missing Coverage**.

The **Add Tests for Missing Coverage** options open.

- 3 Under **Harness**, choose Create a new harness.
- 4 Click **OK** to add tests to the test suite using Simulink Design Verifier.
- 5 Run the updated test suite. On the test manager toolstrip, click **Run**. The test results include coverage for the combined test case inputs, achieving increased model coverage.

## See Also

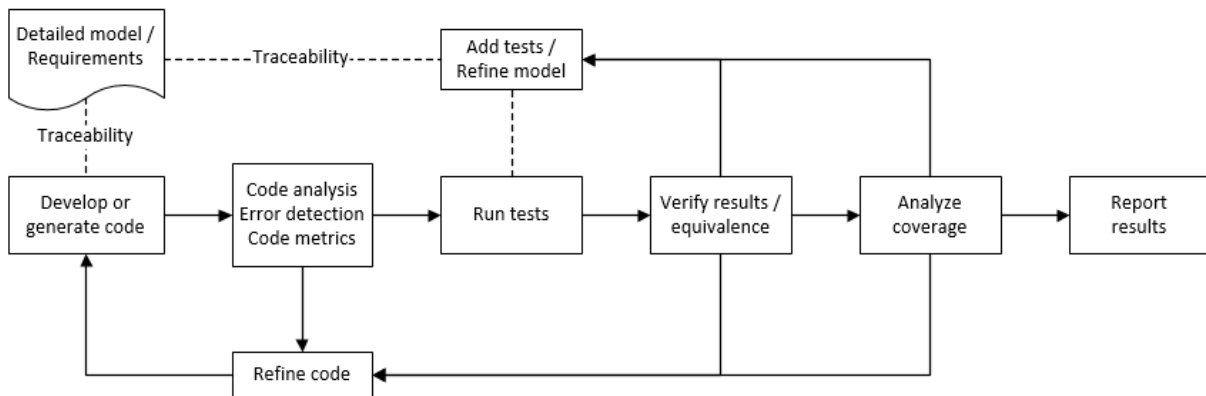
### Related Examples

- “Link Tests to Requirements” on page 1-2
- “Run-Time Assessments” on page 3-49
- “Test Model Output Against a Baseline” on page 6-9
- “Highlight Functional Dependencies” (Simulink Design Verifier)
- “Generate Test Cases for Model Decision Coverage” (Simulink Design Verifier)
- “Increase Test Coverage for a Model”

## Analyze Code and Test Software-in-the-Loop

### Code Analysis and Testing Software-in-the-Loop Overview

Analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. Typically for handwritten code, you check for run-time errors with static code analysis and run test cases that evaluate the code against requirements and evaluate code coverage. Based on the results, refine the code and add tests. For generated code, demonstrate that code execution produces equivalent results to the model by using the same test cases and baseline results. Compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.



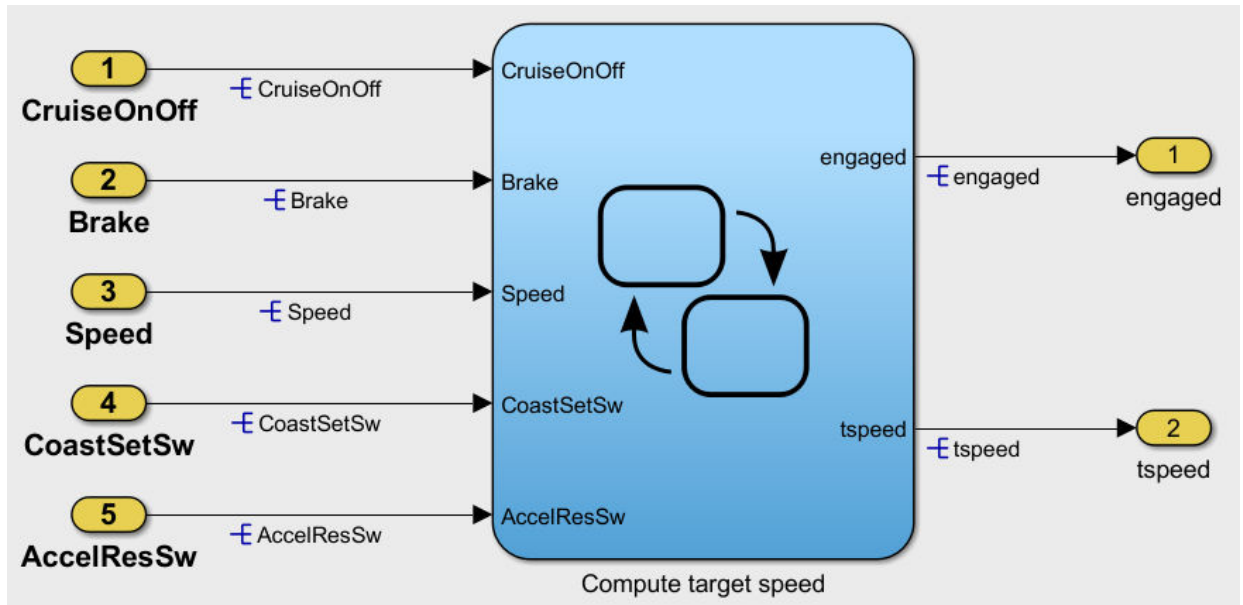
### Analyze Code for Defects, Metrics, and MISRA C:2012

This workflow describes how to check if your model produces MISRA® C:2012 compliant code and how to check your generated code for code metrics, code defects, and MISRA compliance. To produce more MISRA compliant code from your model, you use the code generation and Model Advisor. To check whether the code is MISRA compliant, you use the Polyspace MISRA C:2012 checker and report generation capabilities. For this example, you use the model `simulinkCruiseErrorAndStandardsExample`. To open the model:

- 1 Open the Simulink project:

```
path = fullfile(matlabroot,'toolbox','shared','examples',...
'verification','src','cruise')
run(fullfile(path,'slVerificationCruiseStart'))
```

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

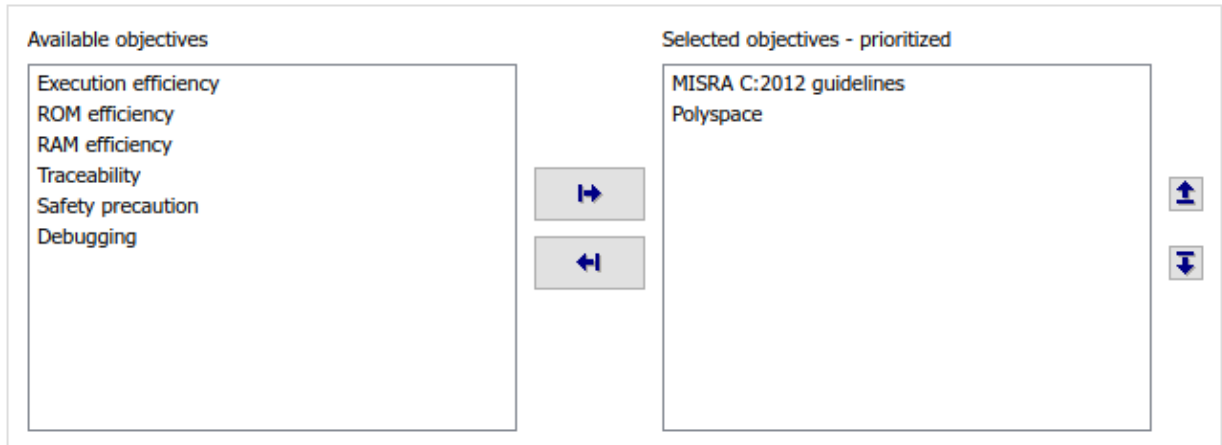


### Run Code Generator Checks

Before you generate code from your model, there are steps that you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows how to use the Code Generation Advisor to check your model before generating code.

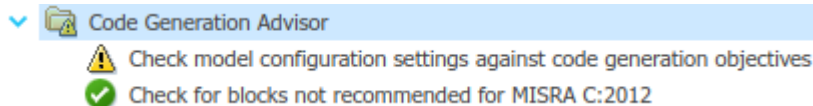
- Right-click `Compute target speed` and select **C/C++ > Code Generation Advisor**.
- Select the Code Generation Advisor folder. Add the Polyspace objective. The MISRA C:2012 guidelines objective is already selected.

Code Generation Objectives (System target file: ert.tlc)



### 3 Click **Run Selected Checks**.

The Code Generation Advisor checks whether there are any blocks or configuration settings that are not recommended for MISRA C:2012 compliance and Polyspace code analysis. For this mode, the check for incompatible blocks passes, but there are some configuration settings that are incompatible with MISRA compliance and Polyspace checking.



### 4 Click on check that was not passed. Accept the parameter changes by selecting **Modify Parameters**.

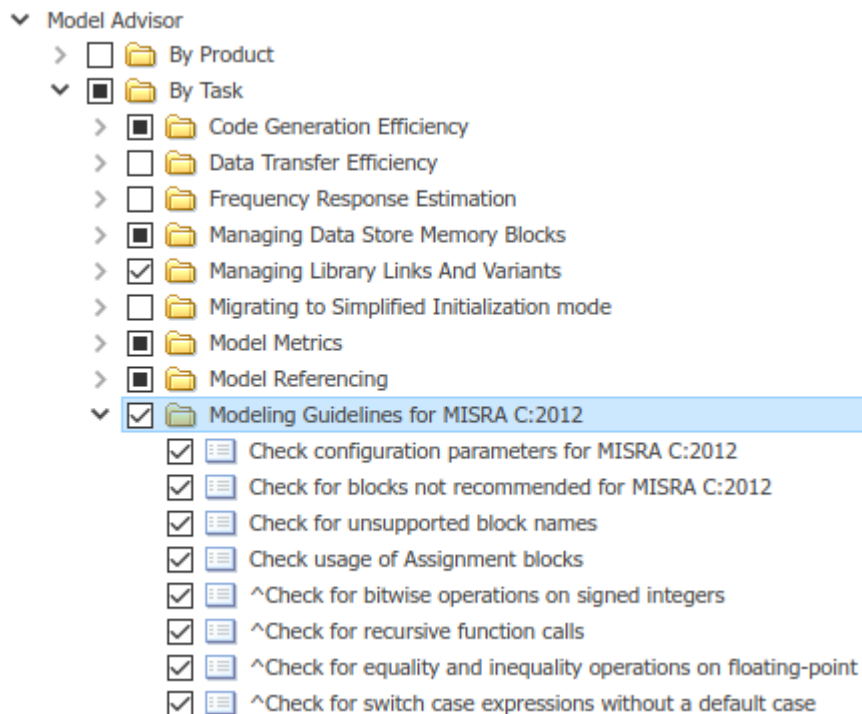
### 5 Rerun the check by selecting **Run This Check**.

## Run Model Advisor Checks

Before you generate code from your model, there are steps you can take to generate code more compliant with MISRA C and more compatible with Polyspace. This example shows you how to use the Model Advisor to check your model further before generating code.

For more checking before generating code, you can also run the Modeling Guidelines for MISRA C:2012.

- 1 At the bottom of the Code Generation Advisor window, select **Model Advisor**.
- 2 Under the **By Task** folder, select the **Modeling Guidelines for MISRA C:2012** advisor checks.



- 3 Click **Run Selected Checks** and review the results.
- 4 If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

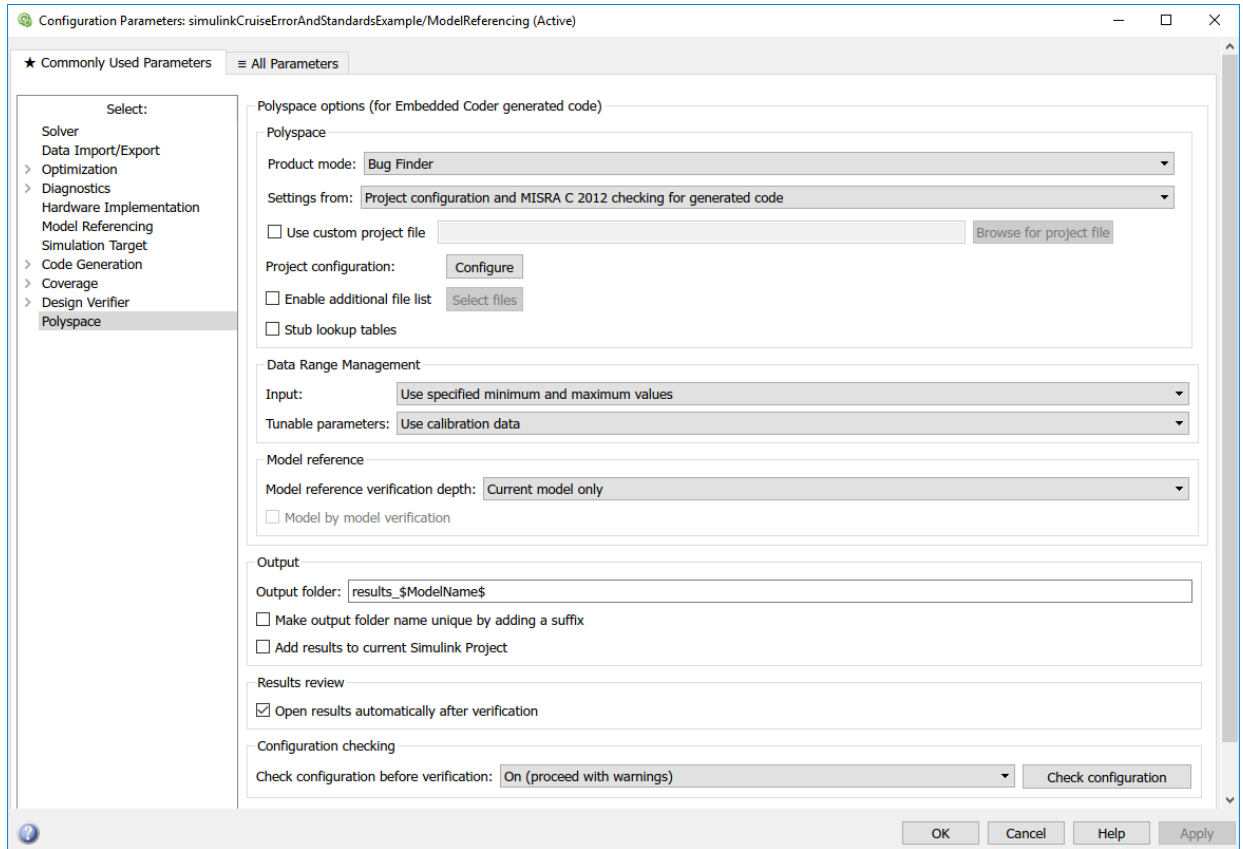
### Generate and Analyze Code

After you have done the model compliance checking, you can now generate code. With Polyspace, you can check your code for compliance with MISRA C:2012 and generate reports to demonstrate compliance with MISRA C:2012.

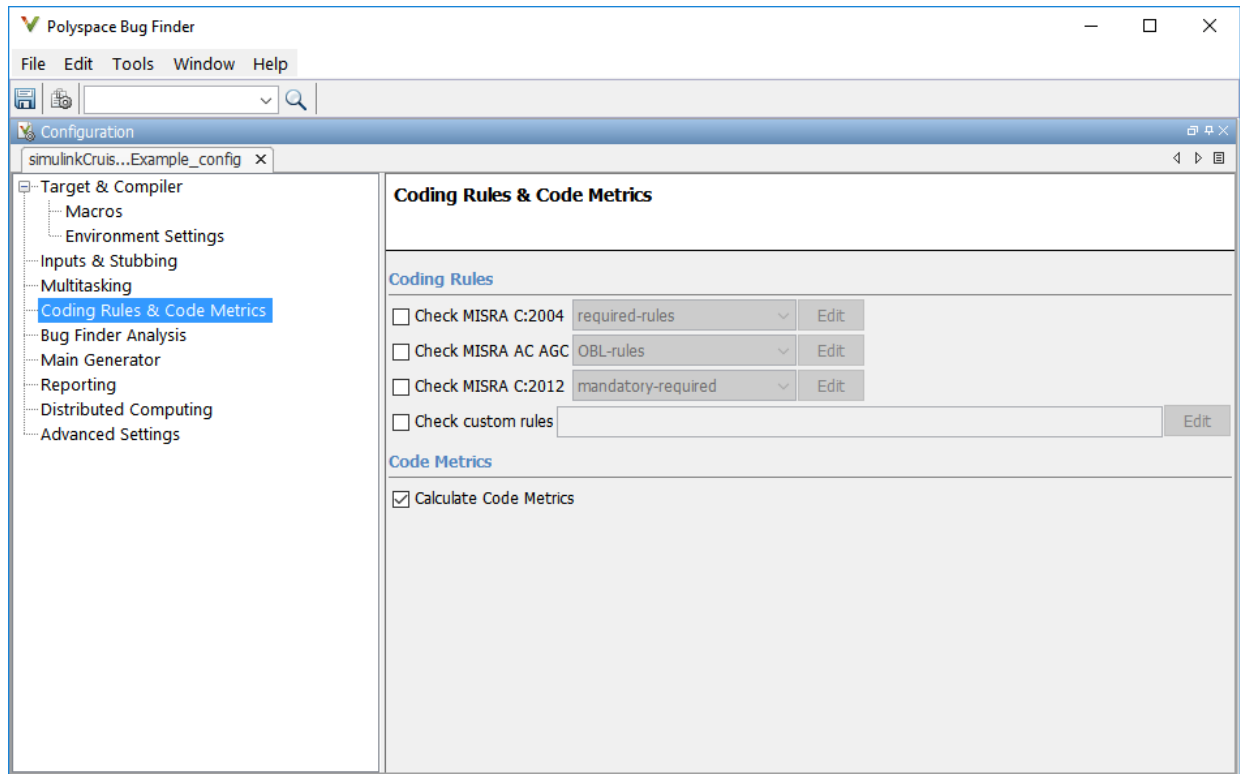
- 1 In the Simulink editor, right-click Compute target speed and select **C/C++ > Build This Subsystem**.
- 2 Use the default settings for the tunable parameters and select **Build**.



- 3 After the code is generated, right-click Compute target speed and select **Polyspace > Options**.



- 4 Click the **Configure** (Polyspace Bug Finder) button. This option allows you to choose more advanced Polyspace analysis options in the Polyspace configuration window.



- 5 On the same pane, select **Calculate Code Metrics**. This option turns on code metric calculations for your generated code.
- 6 Save and close the Polyspace configuration window.
- 7 From your model, right-click Compute target speed and select **Polyspace > Verify Code Generated For > Selected Subsystem**.

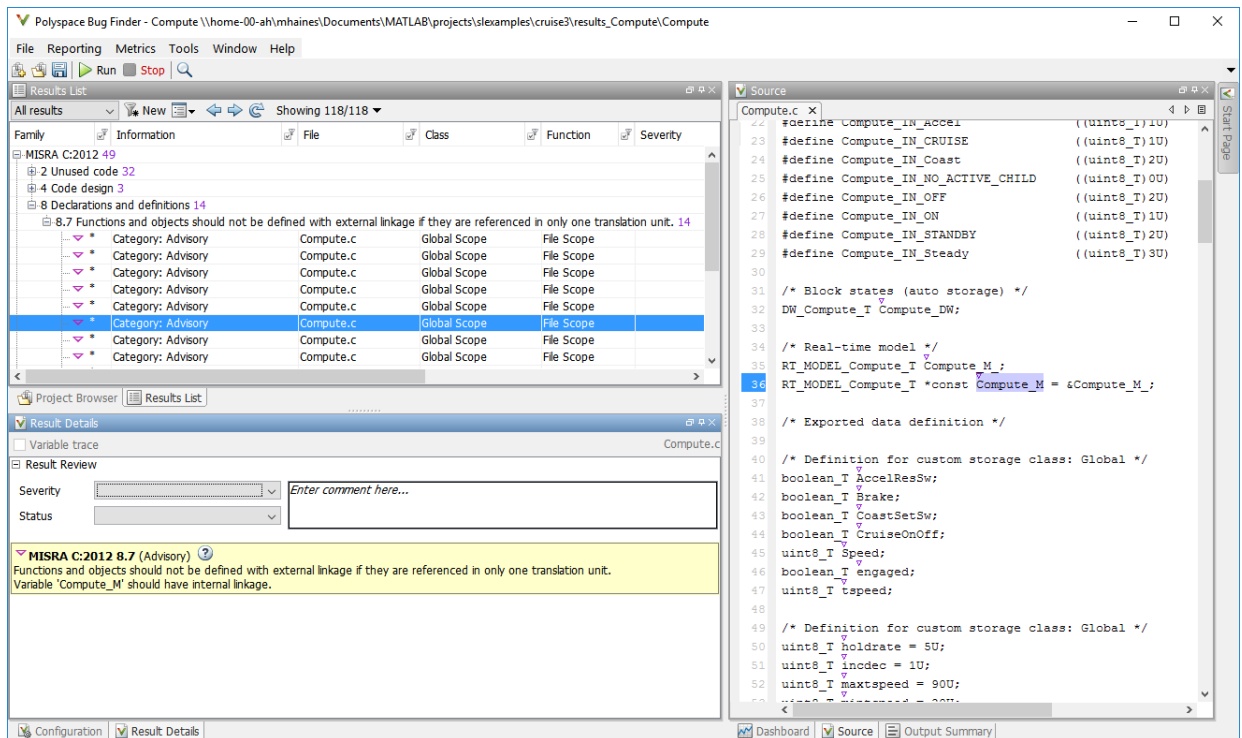
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks and defect checks. You can see the progress of the analysis in the MATLAB Command Window. Once the analysis is finished, the Polyspace environment opens.

### Review Results

After you run a Polyspace analysis of your generated code, the Polyspace environment shows you the results of the static code analysis. There are 50 MISRA C:2012 coding rule violations in your generated code.

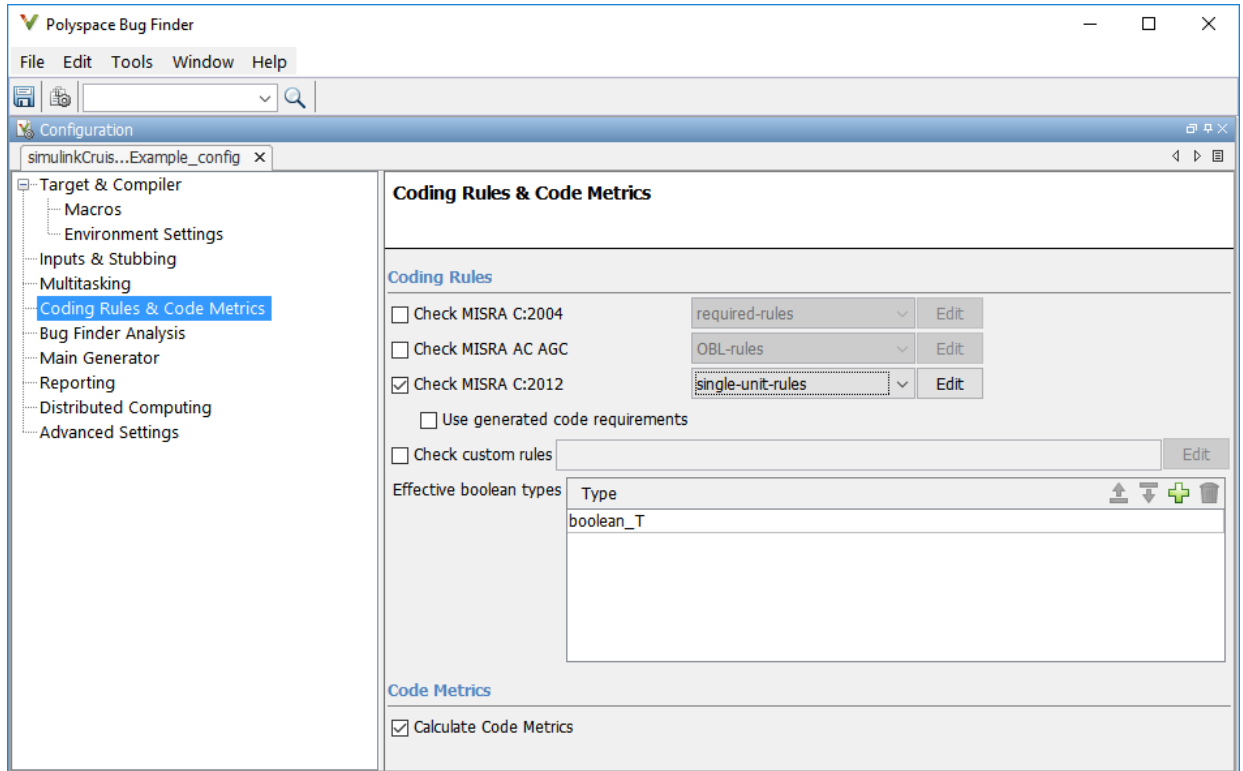
- 1 Expand the tree for rule 8.7 and click through the different results.

Rule 8.7 states that functions and objects should not be global if the function or object is local. As you click through the 8.7 violations, you can see that these results refer to variables that other components also use, such as `CruiseOnOff`. You can annotate your code or your model to justify every result. But, because this model is a unit in a larger program, you can also change the configuration of the analysis to check only a subset of MISRA rules.



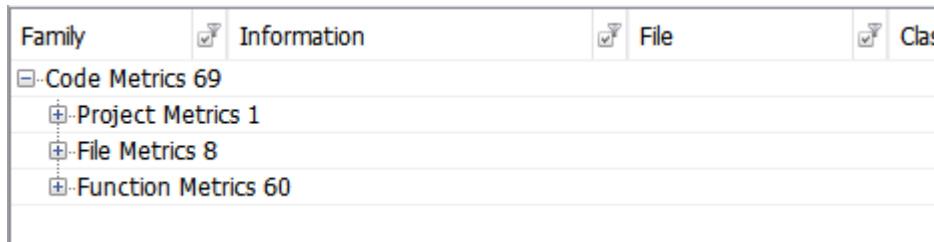
- 2 In your model, right-click Compute target speed and select **Polyspace > Options**.
- 3 Set the **Settings from** (Polyspace Bug Finder) option to **Project configuration**. This option allows you to choose a subset of MISRA rules in the Polyspace configuration.
- 4 Click the **Configure** button.
- 5 In the Polyspace Configuration window, on the **Coding Rules & Code Metrics** pane, select the check box **Check MISRA C:2012** and from the drop-down list, select

single-unit-rules. Now, Polyspace checks only the MISRA C:2012 rules that are applicable to a single unit.



- 6 Save and close the Polyspace configuration window.
- 7 Rerun the analysis with the new configuration.

When the Polyspace environment reopens, there are no MISRA results, only code metric results. The rules Polyspace showed previously were found because the model was analyzed by itself. When you limited the rules Polyspace checked to the single-unit subset, no violations were found.



When this model is integrated with its parent model, you can add the rest of the MISRA C:2012 rules.

### Generate Report

To demonstrate compliance with MISRA C:2012 and report on your generated code metrics, you must export your results. This section shows you how to generate a report after the analysis. If you want to generate a report every time you run an analysis, see [Generate report](#).

- 1 If they are not open already, open your results in the Polyspace environment.
- 2 From the toolbar, select **Reporting > Run Report**.
- 3 Select **BugFinderSummary** as your report type.
- 4 Click **Run Report**.

The report is saved in the same folder as your results.

- 5 To open the report, select **Reporting > Open Report**.

## See Also

### Related Examples

- “Run Polyspace Analysis on Code Generated with Embedded Coder” (Polyspace Bug Finder)
- “Test Two Simulations for Equivalence”
- “Export Test Results and Generate Reports” on page 7-9

